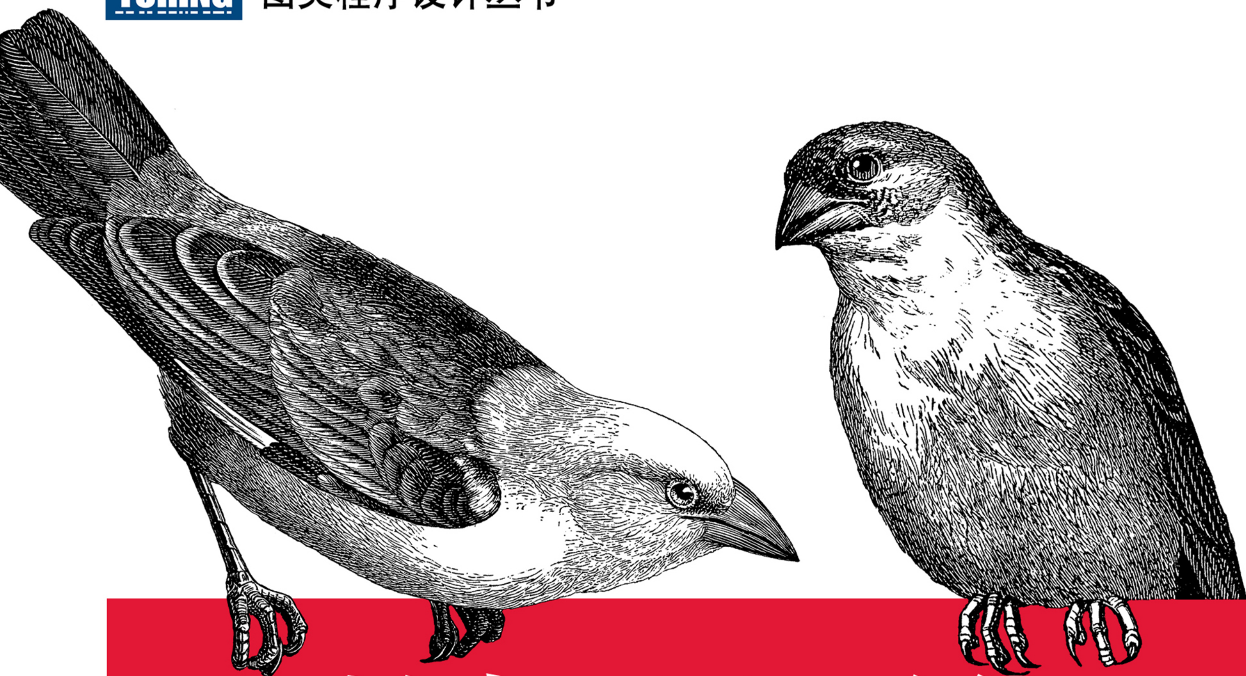
O'REILLY®

TURING 图灵程序设计丛书



大数据项目管理

从规划到实现

Foundations for Architecting Data Solutions

大数据项目的“孙子兵法”，助你拥有软件开发大局观

[美] 特德·马拉斯卡　乔纳森·塞德曼 著

薛命灯 译

 中国工信出版集团　 人民邮电出版社 POSTS & TELECOM PRESS

译者介绍

薛命灯

InfoQ高级社区编辑，毕业于厦门大学软件学院，拥有十余年软件开发和架构经验，曾在多家大型软件公司任职，另译有《Kafka权威指南》等技术图书。

数字版权声明



图灵程序设计丛书

大数据项目管理：从规划到实现

Foundations for Architecting Data Solutions: Managing Successful Data Projects

[美] 特德 • 马拉斯卡 [美] 乔纳森 • 塞德曼 著
薛命灯 译

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

O'Reilly Media, Inc.授权人民邮电出版社出版

人 民 邮 电 出 版 社
北　　京

图书在版编目（CIP）数据

大数据项目管理 : 从规划到实现 / (美) 特德·马拉斯卡 (Ted Malaska) , (美) 乔纳森·塞德曼 (Jonathan Seidman) 著 ; 薛命灯译. -- 北京 : 人民邮电出版社, 2020.1
（图灵程序设计丛书）
ISBN 978-7-115-45736-3

Ⅰ. ①大… Ⅱ. ①特… ②乔… ③薛… Ⅲ. ①数据处理—研究 Ⅳ. ①TP274

中国版本图书馆CIP数据核字(2019)第270722号

内 容 提 要

本书提供了一个框架，从整体上介绍与大数据项目开发相关的基本概念，帮助读者评估大数据项目，理解成功的现代数据项目的基本要素。全书共 8 章，内容包括现代数据项目的主要类型、生命周期、风险管理、接口设计、分布式存储系统、元数据管理、数据处理等。本书旨在让读者厘清思路，顺利地从数据项目的规划阶段走到执行阶段，实现健壮、可维护的架构和解决方案。

本书适合首席信息官、首席运营官、技术主管、系统架构师及相关的开发人员阅读。

◆ 著　[美] 特德·马拉斯卡 [美] 乔纳森·塞德曼
译　薛命灯
责任编辑　谢婷婷
责任印制　周昇亮

◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路11号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　印刷

◆ 开本：800×1000　1/16
印张：9.75
字数：231千字　2020年 1 月第 1 版
印数：1 – 3 000册　2020年 1 月北京第 1 次印刷

著作权合同登记号　图字：01-2019-6607号

定价：59.00元

读者服务热线：(010)51095183转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广登字 20170147 号

版权声明

O’Reilly Media, Inc.介绍

O’Reilly 以“分享创新知识、改变世界”为己任。40 多年来我们一直向企业、个人提供成功所必需之技能及思想，激励他们创新并做得更好。

O’Reilly 业务的核心是独特的专家及创新者网络，众多专家及创新者通过我们分享知识。我们的在线学习（Online Learning）平台提供独家的直播培训、图书及视频，使客户更容易获取业务成功所需的专业知识。几十年来 O’Reilly 图书一直被视为学习开创未来之技术的权威资料。我们每年举办的诸多会议是活跃的技术聚会场所，来自各领域的专业人士在此建立联系，讨论最佳实践并发现可能影响技术行业未来的新趋势。

我们的客户渴望做出推动世界前进的创新之举，我们希望能助他们一臂之力。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列非凡想法（真希望当初我也想到了）建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的领域，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，那就走小路。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

前言 ix

第 1 章　数据项目的主要类型及考虑因素 1

1.1　数据项目的主要类型 1

1.2　数据管道和数据暂存 3

1.2.1　主要考虑因素和风险管理 4

1.2.2　数据管道和数据暂存团队的人员组成 13

1.3　数据的处理和分析 14

1.3.1　主要考虑因素和风险管理 14

1.3.2　数据处理和分析团队的人员组成 17

1.4　应用程序开发 17

1.4.1　主要考虑因素和风险管理 18

1.4.2　应用程序开发团队的人员组成 22

1.5　小结 22

第 2 章　评估和选择数据管理解决方案 25

2.1　开源项目的阶段 26

2.1.1　孵化阶段 27

2.1.2　发布阶段 27

2.1.3　“治愈癌症”阶段 27

2.1.4　打破承诺阶段 28

2.1.5　强化阶段 29

2.1.6　企业阶段 30

2.1.7　终结阶段 30

2.2　开源项目的常见生命周期 31

2.2.1 使产品起死回生 32
2.2.2 追随者 33
2.3 评估基准测试 34
2.4 技术选型的考虑因素 35
2.4.1 了解构建块 36
2.4.2 寻求建议 37
2.4.3 从分析师那里获得见解 37
2.4.4 研究市场趋势 37
2.5 小结 39

第 3 章 数据项目的风险管理 41

3.1 风险类型 41
3.1.1 技术风险 41
3.1.2 团队风险 42
3.1.3 需求风险 42
3.2 风险管理 42
3.2.1 对架构中的风险进行分类 42
3.2.2 技术风险 45
3.2.3 团队的优势 45
3.2.4 外部团队风险 47
3.2.5 需求风险 47
3.2.6 融会贯通 47
3.3 使用原型和 PoC 50
3.3.1 找到两三种方法 50
3.3.2 进行 PoC，然后丢弃 50
3.3.3 部署的注意事项 50
3.4 使用接口 51
3.5 尽早开始构建 52
3.6 频繁测试并保留记录 52
3.7 监控和警报 53
3.8 沟通风险 54
3.8.1 合作并获得信任 54
3.8.2 公开风险 54
3.9 将风险作为谈判工具 55
3.10 小结 55

第 4 章 接口设计 57

4.1 人体 57
4.1.1 人体与数据架构 57
4.1.2 解耦 61

4.1.3 解耦的注意事项 63
4.1.4 专门化 64
4.2 什么造就了好的接口设计 64
4.2.1 合约 64
4.2.2 抽象 64
4.2.3 版本控制 65
4.2.4 防御 65
4.2.5 接口的文档和命名 66
4.3 非功能性考虑因素 67
4.3.1 可用性 67
4.3.2 响应时间 68
4.3.3 负载容量 68
4.3.4 使用测试来确定 SLA 69
4.4 通用接口示例 69
4.4.1 发布 – 订阅 69
4.4.2 异步请求 – 响应 71
4.4.3 同步请求 – 响应 72
4.5 小结 73
第 5 章 分布式存储系统 75
5.1 分布式存储系统的属性 75
5.1.1 谱系 76
5.1.2 分区 77
5.1.3 处理数据变更 78
5.1.4 读取路径 80
5.1.5 可用性与一致性 84
5.1.6 主要用例 85
5.2 存储系统细分 85
5.2.1 HDFS 86
5.2.2 S3 和对象存储系统 87
5.2.3 Apache HBase 89
5.2.4 Apache Cassandra 90
5.2.5 Elasticsearch 和 Apache Solr 94
5.2.6 新进者：Apache Kudu 和 CockroachDB 95
5.2.7 内存存储系统 96
5.3 小结 99
第 6 章 企业元数据 101
6.1 为什么要关注元数据 102
6.1.1 数据可见性 102

6.1.2 数据之间的关系 103
6.1.3 数据监管 104
6.2 数据架构中的元数据类型 105
6.2.1 静态数据 106
6.2.2 动态数据 107
6.2.3 数据源的元数据 110
6.2.4 有关数据处理的元数据 111
6.2.5 报告和仪表盘 112
6.3 元数据收集 112
6.3.1 声明式元数据收集 113
6.3.2 发现式元数据收集 114
6.4 元数据管理实践 115
6.5 小结 116
第 7 章 确保数据完整性 117
7.1 构建数据管道 118
7.2 验证数据管道 123
7.2.1 行数 123
7.2.2 唯一计数 124
7.2.3 全字节比较 124
7.2.4 校验和比较 125
7.3 小结 126
第 8 章 数据处理 127
8.1 处理引擎的属性 127
8.1.1 DAG 管理 128
8.1.2 计算隔离 130
8.1.3 性能 132
8.1.4 容错 132
8.1.5 交互模型 135
8.1.6 批处理和流处理 135
8.2 数据处理演变史 136
8.3 小结 138
关于作者 139
关于封面 139

前言

既然你开始阅读本书，那么就应该知道，近几年来，数据管理领域发生了巨大的变化。我们已经看到了从第三方专有解决方案到新的开源分布式数据系统的转变。通常使用“大数据”来指代这些新的解决方案（我们发现这个词的指代作用越来越弱），但其实早期的很多专有系统也采用了可以存储和处理大量数据的分布式架构。尽管这些专有解决方案和新的开源解决方案都可以用来解决很多相同的问题，但它们之间存在一些明显的差异，这些差异促成了新系统的发展。这些差异不仅体现在开源的经济性方面，也与技术的发展有关。技术的发展促进了新系统的实现，而如果使用以前的解决方案来实现这些系统颇具挑战性。

随着这些系统的发展，出现了很多相关的书、文章、培训、会议等。这些资源可以帮助你以及这个领域的其他从业者更好地使用这些系统。那么，为什么还要再写一本与“大数据”相关的书呢？我们想说的是：不要因为一棵树而错过整片森林。这些资源大都侧重于底层的细节，例如使用 MapReduce 或 Spark 之类的分布式处理引擎来实现应用程序，或者应用高级算法来分析数据。除此之外，也有一些资源关注更高层次的架构，例如由本书作者和另外两位作者合著的《Hadoop 应用架构》[1]。

这些资源缺乏的是一个更广阔的视野，换句话说，需要采取哪些步骤来确保数据项目能够从规划阶段成功地走到执行阶段？要成功地实施数据项目，获取与架构和组件系统相关的专业知识固然重要，但其他的一些考虑因素也同样重要，而这些因素往往在探索新技术的过程中被忽视。

这些考虑因素包括：

- 理解问题；
- 选择适合用例的软件解决方案；

注 1：《Hadoop 应用架构》由人民邮电出版社出版，详见 http://ituring.cn/book/1710。——编者注

- 应对项目风险；
- 组建团队，以便成功交付项目；
- 在项目进行过程中，实现健壮、可维护的架构和解决方案。

如果你是经验丰富的软件开发人员，可能已经很熟悉这些因素了。成功管理现代数据项目的大部分流程与管理其他软件开发项目是一样的，只是在开发新的软件系统和架构时，需要一些新的知识，还需要考虑到一些额外的事项。例如，评估开源软件与选择专有解决方案有很大的不同。我们的目的不是提供又一本有关软件项目管理的书，而是指导你将行之有效的项目管理和开发实践应用到现代数据解决方案中。

读者对象

本书主要面向数据项目的决策者和实施者，例如以下角色：

- 负责高层决策的首席信息官或首席技术官；
- 负责交付数据项目的项目经理和产品经理；
- 负责开发数据项目的首席架构师、技术主管和开发人员。

再次强调，我们不打算介绍如何使用特定组件来实现应用程序；相反，我们会提供一个框架，帮助你理解成功的现代数据项目都有哪些基本要素。我们希望你能够掌握这些知识，从而成功地掌控数据项目，并做出正确的项目决策，让项目为用户带来真正的价值。

阅读方式

本书的每一章都会涉及一个与数据项目管理相关的主题。你不必从头到尾阅读整本书，因为大多数章节的内容相对独立。不过，在启动数据项目之前，先阅读第 1~3 章将大有裨益。

以下是各章的主要内容。

第 1 章，*数据项目的主要类型及考虑因素*，概述 3 种主要的数据项目用例，并针对每个用例列举需要注意的一系列考虑因素。在启动新的数据项目之前，最好先阅读这一章。

第 2 章，*评估和选择数据管理解决方案*，为在分布式开源世界中选择技术解决方案提供指导。如果你正尝试启动数据项目，或者刚刚进入这个领域，这一章对于你来说也会非常有用。

第 3 章，*数据项目的风险管理*，讨论项目风险以及如何管理它们。风险管理是软件项目的一项重要活动，大型数据项目存在一些独特的风险，要成功实现这些项目，需要管理好它们。

第 4 章，*接口设计*，讨论系统接口的设计和实现。对于创建可维护和可扩展的系统来说，定义有效的抽象和合约至关重要。因此，我们在这一章会根据自己实现大型数据项目的经验提供一些指导。

第 5 章，*分布式存储系统*，讨论分布式存储系统。数据存储是所有数据系统的核心组件，这一章将列举一些常用的分布式存储系统。更重要的是，它还会提供一个用于评估存储系统的框架。

第 6 章，*企业元数据*，讨论元数据管理。这是在构建数据系统时的另一个至关重要但经常被忽视的方面。

第 7 章，*确保数据完整性*，讨论数据的完整性问题。这是在构建数据系统时的另一个需要注意的事项，需要在项目开始时进行规划。在构建支持多种存储格式的数据系统时，确保数据的完整性和传承关系变得更具挑战性。

第 8 章，*数据处理*，讨论可用于处理分布式数据的框架。在构建有价值的数据系统时，处理和分析数据的能力是另一个重要方面。与第 5 章类似，这一章也会提供一个框架，用于了解可用的数据处理系统以及评估哪些系统适合你的应用场景。

排版约定

本书使用下列排版约定。

- **黑体字**
  表示新术语或重点强调的内容。

- 等宽字体（`constant width`）
  表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。



该图标表示一般注记。

使用代码示例

本书是要帮你完成工作的。一般来说，如果本书提供了示例代码，你可以把它用在你的程序或文档中。除非你使用了很大一部分代码，否则无须联系我们获得许可。比如，用本书的几个代码片段写一个程序就无须获得许可，销售或分发 O’Reilly 图书的示例光盘则需要获得许可；引用本书中的示例代码回答问题无须获得许可，将书中大量的代码放到你的产品文档中则需要获得许可。

我们很希望但并不强制要求你在引用本书内容时加上引用说明。引用说明一般包括书名、作者、出版社和 ISBN，例如“*Foundations for Architecting Data Solutions* by Ted Malaska

and Jonathan Seidman (O'Reilly). Copyright 2018 Ted Malaska and Jonathan Seidman, 978-1-492-03874-0"。

如果你觉得自己对示例代码的用法超出了上述许可的范围，欢迎你通过 permissions@oreilly.com 与我们联系。

O'Reilly Safari

Safari（之前称作 Safari Books Online）是一个针对企业、政府、教育者和个人的会员制培训和参考平台。

会员可以访问来自 250 多家出版商的上千种图书、培训视频、学习路径、互动式教程和精选播放列表，这些出版商包括 O'Reilly Media、Harvard Business Review、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Adobe、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 等。

要了解更多信息，可以访问 http://www.oreilly.com/safari。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表[2]、示例代码以及其他信息。本书的网站地址是：http://shop.oreilly.com/product/0636920161417.do。

对于本书的评论和技术性问题，请发送电子邮件到 bookquestions@oreilly.com。

要了解更多 O'Reilly 图书、培训课程、会议和新闻的信息，请访问网站：http://www.oreilly.com。

注 2：本书中文版勘误，请到 http://ituring.cn/book/2641 查看和提交。——编者注

我们在 Facebook 的地址如下：http://facebook.com/oreilly。

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia。

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia。

致谢

在撰写本书的过程中，很多人为我们提供了宝贵的反馈和支持，尤其是 Mark Grover、Kevin O'Dell 和 Steven Totman，他们为审阅本书内容付出了宝贵的时间。这些审阅者帮助我们提高了本书的质量，如果书中仍存在错误，都应该由我们自己负责。

我们要感谢 O'Reilly 的编辑 Nicole Tache 和 Michele Cronin。正是在她们的指导下，我们才顺利完成了本书。我们还要感谢 O'Reilly Media 的其他人提供的帮助和支持。

如果致谢清单遗漏了哪位，我们深表歉意。

电子书

扫描如下二维码，即可购买本书电子版。



第 1 章 数据项目的主要类型及考虑因素

了解自己要构建什么，以及在设计可靠的解决方案时需要考虑哪些主要因素，这些是任何一个数据项目取得成功的基本条件。我们根据经验将数据项目分为 3 种类型，它们代表了大多数的数据项目。这种分类方式有助于在开始实现解决方案之前探究需要考虑的主要因素。并非每个项目都恰好属于其中一个类别，有些项目甚至可能同属多个类别。但我们认为，这些项目类型提供了一个有用的框架，帮你更好地了解数据用例。

本章首先描述主要的项目类型，然后介绍实现解决方案时需要考虑的主要事项，最后深入探讨每种项目类型的考虑因素。

1.1 数据项目的主要类型

我们先来看看数据项目的 3 种主要类型。

- 数据管道和数据暂存

  可以将这类项目视为提取 – 转换 – 加载型项目，换句话说，这类项目涉及对数据集的收集、暂存、存储、建模，等等。实际上，这类项目为执行后续的数据处理和分析奠定了基础。

- 数据的处理和分析

  这类项目最终会提供某种可用价值，可能是生成报告、创建和执行机器学习模型，等等。

- 应用程序开发

  这类项目提供能够实时支持业务需求的数据框架，例如 Web 应用程序或移动应用程序的数据后端。

接下来，本章将着重关注每个项目类型的以下方面。

- 主要考虑因素

  尽管这 3 种项目类型有很多共同点，但也有一些会影响架构决策和优先级的区别，而架构决策将反过来推动项目的其余部分。在深入探讨这 3 种项目类型时，我们将首先详细介绍每个项目类型的主要考虑因素。

- 风险管理

  任何数据项目都伴随着一定的风险。我们将讨论与特定项目类型相关的潜在风险及处理方法。在很多情况下，特定场景的风险会有多种风险管理方法，因此我们需要从不同的维度进行探讨。



第 3 章将详细介绍风险管理。

- 团队组成

  为交付不同类型的项目组建团队时，需要考虑到一系列因素。不同类型的项目所需要的技能、经验和兴趣是不一样的，因此我们就每一种项目类型提供一些用于组建团队的建议。

- 安全

  安全问题可能是所有项目都会涉及的一个重要的考虑因素。安全是一个非常重要和宽泛的主题，所涉及的内容可以单独写成一本书。事实上，针对你所使用的系统，能够找到一些有用的参考资料。因为这是一个非常重要的主题，所以本书不会详细介绍，但会列出在项目过程中需要牢记的一些安全事项。

对于某些开源数据管理系统而言，安全措施更像是马后炮。这是因为早期用户更关心与存储和处理大量数据的能力相关的技术问题。此外，这些系统通常部署在内部网络中，对它们的访问是可控的。随着越来越多的企业部署这些解决方案，他们也越来越关注存储在这些系统中的数据的安全性和私密性。于是，这些项目和供应商努力做出变更和改善，以便帮助企业更好地使用这些系统。

在为项目安全做规划时，应该考虑以下维度。

- 身份验证

  确保访问系统的用户是合法的。任何成熟的系统都应该支持强身份验证，这通常可以通过 Kerberos 或轻量目录访问协议等方式来实现。

- ❑ 授权

  在确保访问用户的合法性之后，还需要决定他们可以访问哪些数据。成熟的系统需要提供不同粒度的访问控制。例如，不仅可以提供数据库表级别的访问控制，还可以提供列级别的访问控制。在为敏感数据构建数据架构时，具备控制哪些用户和用户组可以访问哪些特定数据的能力是非常重要的。

- ❑ 加密

  除了控制对数据的访问，出于安全方面的考虑，保护这些数据免受恶意用户和恶意入侵的影响也至关重要。数据加密是最常用的保护方法。我们需要从两个角度来考虑这个问题。

  - **静止的数据**是指已经进入系统并保存在磁盘上的数据。很多数据管理供应商为此提供了解决方案，并将它们作为管理平台的一部分。一些第三方供应商也为此提供了解决方案。
  - **传输中的数据**是指在系统中移动的数据。通常，供应商或项目会为此提供标准的加密机制，例如传输层安全协议。

- ❑ 审计

  安全问题的最后一个考量维度是能够捕获与数据相关的活动，比如数据的传承关系、谁在访问数据，以及如何使用数据，等等。这个问题仍然需要通过供应商或项目提供的工具来解决。

如果安全对项目非常重要，最好的办法是找到可以解决上述 4 个问题的方案或供应商。这样一来，就可以减少花在数据安全性管理方面的时间，而将更多的时间用于解决其他问题。

1.2 数据管道和数据暂存

我们从 3 个数据项目类型中范围最广的开始讨论，因为它涉及从外部数据源到目标数据源的整个路径，并为构建数据解决方案的其余部分奠定了基础。

对于这个项目类型，在设计解决方案时需要考虑以下因素：

- 针对目标数据将执行哪些类型的查询和处理；
- 客户的数据要求；
- 已收集数据的类型。

考虑到这些数据在后续处理和分析中的重要性，我们在建模和存储这些数据时要十分谨慎，为后续的数据访问提供便利。

1.2.1 主要考虑因素和风险管理

对于数据管道和数据暂存项目，有以下主要考虑因素：

- 源数据消费；
- 数据传递保证；
- 数据的管理和治理；
- 延迟和传递确认；
- 目标数据的访问模式。

接下来，我们将逐个介绍这些考虑因素以及每个因素的属性会如何影响项目的优先级。

1. 源数据消费

当我们说到数据源时，基本上是指那些生成数据的系统，它们为我们构建的数据解决方案提供必要的数据。数据源可以是手机、传感器、应用程序、机器日志、操作型数据库和事务型数据库，等等。数据源大都位于数据管道和数据暂存系统之外。实际上，你可以根据花费在与数据源团队合作上的时间来评估系统的成功程度。数据工程团队在数据源集成上花费的时间通常与数据源集成设计的优劣成反比。

可以使用一些标准的方法收集源数据。

❑ 嵌入式代码

你可以为源系统提供代码，将它们嵌入到源系统中，这些代码知道如何将必要的数据发送到你的数据管道中。

❑ 代理

这是一个非常靠近数据源的独立系统，大多数情况下与数据源位于同一设备上。与嵌入式代码不同，代理是作为单独的进程运行的，而且没有依赖项。

❑ 接口

这是最轻量级的方式，例如 REST 和用于接收源数据的 WebSocket 端点。

当然，除了这些，还有其他一些常用的数据收集方式：

- 第三方数据集成工具，可以是开源的，也可以是商用的；
- 批量数据摄取工具，例如 Apache Sqoop 和特定项目提供的工具（如 Hadoop 分布式文件系统提供的 `put` 命令）。

你可以根据实际的用例选择工具，它们可以帮你更好地构建数据管道。因为其他参考资料以及供应商和项目的文档已经详细介绍了它们，所以本书不再赘述。

哪种方法最好？答案通常取决于数据来源。但在某些情况下，可能几种方法都适用，关键

是要确保正确地使用这些方法。因此，我们将讨论与不同数据收集类型相关的一些注意事项。先从嵌入式代码开始讲。

嵌入式代码

在使用嵌入式代码收集源数据时，需要考虑以下准则。

- 限制编程语言的使用

  不要试图支持多种编程语言，而应该先使用一种语言实现，然后为其他语言提供绑定。例如，使用 C、C++ 或 Java 实现，然后为需要支持的其他语言创建绑定。以 Kafka 为例，Kafka 的核心项目提供了 Java 版本的生产者和消费者，而其他语言的库或客户端需要绑定到 Kafka 提供的库，这些库是 Kafka 发行包的一部分。

- 限制依赖项的使用

  任何嵌入式代码都存在潜在的库冲突。限制依赖项的使用有助于缓解这个问题。

- 提供可见性

  人们可能会关注嵌入式代码中究竟包含了哪些内容，所以需要通过开源或将代码放在公开代码库中来提供嵌入式代码的可见性，这是一种简单而安全的方式。用户可以看到所有的代码，进而减轻对某些潜在问题（如内存使用、网络使用等）的担忧。

- 运维问题

  还有一个考虑因素是嵌入式代码可能会在生产环境中造成哪些问题。确保你已经考虑到了内存泄漏或性能问题，并定义了用于解决这些问题的支持模型。日志和代码插桩有助于在发生故障时找出问题。

- 版本管理

  在使用嵌入式代码时，你可能无法控制代码的更新。这个时候，确保向后兼容并定义良好的版本就显得非常重要。

代理

在架构中使用代理时，请注意以下事项。

- 部署

  与架构中的其他组件一样，请确保代理的部署是经过测试的，并且是可重复的。这可能需要使用某种自动化工具或容器。

- 资源使用情况

  确保源系统拥有足够的资源来支持代理进程的运行，包括内存、CPU 等。

- 隔离

  虽然代理在应用程序外部运行，但仍然需要防止代理对数据收集带来负面影响。

- 调试

  同样，当生产环境出现问题时，需要采取调试措施，使系统恢复正常。这可能需要记录日志、进行代码插桩，等等。

接口

以下是使用接口时需要注意的一些事项。

- 版本

  尽管没有嵌入式代码那么令人头疼，但在使用接口时，版本控制仍然是个问题。请确保从一开始就将版本控制作为一个核心考虑因素。

- 性能

  对于任何源数据收集框架来说，性能和吞吐量都是至关重要的。此外，即使自己设计和实现代码来保证性能，你也会发现，数据源或数据接收方的实现可能并不理想。因为你可能无法控制这些代码，所以关键是要能够在出现性能问题时检测到并发出警报。

- 安全

  在代理模型和嵌入式模型中，你可以控制代码，但在接口模型中，接口是唯一的入口屏障。关键是在提供安全性的同时保持接口简单。对于这种情况，有多种模型可用，例如使用安全令牌。

2. 针对源数据消费的风险管理

在构建数据收集系统时，你面临的风险与构建外部 API（application program interface，应用程序接口）面临的风险是一样的，而且包括扩展性问题。下面列出需要注意的一些主要问题。

版本管理

每个人都喜欢能够正常运行的 API。问题是，我们很少能够富有远见地设计接口，使它们在未来不必因为兼容性问题而变更。你需要一个强大的版本控制策略和一个提供向后兼容性保证的计划来应对这种情况，并将这个计划作为沟通策略的一部分。

数据源故障带来的影响

需要为数据源系统的多种故障场景制定应对方案。如果嵌入式代码是数据源执行过程的一部分，那么代码执行失败会导致整个数据收集过程失败。如果你没有使用嵌入式代码，而是使用了其他收集机制（比如代理），那么当它们出现故障时，数据源会受到怎样的影响？数据会丢失吗？这会影响应用程序的预期正常运行时间吗？

为了回答上述问题，你需要了解数据源，找到更多的解决方案并与他人交流，而且要清楚地认识到，发生故障和中断在所难免。这样一来，你就可以在故障发生时提供各种保护。

请注意，对于经过精心设计和实现的数据管道来说，故障应该是很罕见的，但仍然不可避

免。所以，我们需要为数据管道提供适当的机制，使之可以在发生非预期事件时通知我们。例如，监控吞吐量，当吞吐量指标偏离特定的阈值时就发出警报。我们的想法是构建最具弹性的数据管道，在出现问题时知道如何应对。

另外，可以考虑使用管道副本。在发生故障时，如果一个管道瘫痪，另一个管道可以接管。这不只是节点故障保护，拥有另一个单独的管道可以让你免受难以预测的故障的影响（如错误配置的部署或错误的构建）。理想情况下，在设计数据管道时，应该做到像部署Web 应用程序那样部署数据管道。

避免不良数据源的影响

在构建数据摄取系统时，数据源可能会滥用你的 API、发送过多的数据，等等。这些行为都有可能对你的系统产生负面影响。在设计和实现系统时，需要采取措施来防范这些风险。请考虑以下这些注意事项。

- 节流

  这将限制数据源发送给你的记录数量。当数据源向你发送更多的记录时，你的系统可以延长用于接收这些数据的时间。另外，你也可以发出消息告知数据源建立了太多的连接。

- 丢弃

  如果你的系统不需要提供数据保证，那么可以在发生过载或无法处理输入数据的情况下将消息丢弃。不过，一旦这样做了，会给人们留下你的系统会丢失数据的印象，降低系统的整体可信度。但在大多数情况下，只要实时地向数据源告知发生了数据丢弃，让客户端采取适当的行动，那么丢弃数据或许是可以接受的。简单地说，在采用丢弃数据的方案时，请确保客户端知道发生了什么以及如何应对。

3. 数据传递保证

在规划数据管道时，你需要向数据使用者做出很多承诺。对于不同的数据收集系统，可以提供不同级别的数据传递保证。

- 尽力而为

  数据源向你发送一条消息，你尝试传递这条消息，但可能会发生数据丢失。如果不必捕获每一个事件，那么这种情况是可以接受的。例如，通过处理传入的数据来捕获聚合度量指标，但不要求整体的准确性。

- 至少一次

  数据源向你发送一条消息，你复制了这条消息，不希望发生数据丢失。这可能是最常见的情况。虽然它增加了额外的复杂性，但你在大多数情况下可能希望捕获到所有的事件。请注意，这可能需要在管道中添加数据去重的逻辑，但在大多数情况下它比下面描述的“恰好一次”传递保证更容易、成本更低。

- 恰好一次

  数据管道收到一条消息，你确保处理了这条消息，并且不会重复处理。如前所述，这是成本最高的一种保证机制，而且从技术角度来讲也是最复杂的。现在有很多系统都承诺提供这个功能，但你应该仔细考虑是否有必要使用，或者是否可以采用其他现成的机制来处理潜在的数据重复问题。

在大多数情况下，至少一次传递保证已经足够了，因为在摄取数据后进行数据去重的成本通常不会很高。但无论要达到哪一种保证级别，你都应该计划好，将其记录下来并传达给系统用户。

4. 数据的管理和治理

如今，强大的数据收集系统必须具备两个关键特性。

- 数据模型管理

  有能力修改或添加数据模型。

- 数据治理

  有能力了解收集到的数据以及滥用和泄露数据可能带来的风险。



第 6 章将介绍元数据管理，到时候将详细讨论与数据的管理和治理相关的内容。现在，我们先讨论它的范围和目标。

数据模型管理

需要通过适当的机制来捕获系统的数据模型。在理想情况下，数据管道用户不需要依赖你的团队来添加新数据源或更改现有数据源。Kafka 的 Confluent Schema Registry 就是这方面的一个例子，它可用于存储 schema，包括多个版本的 schema。这为不同版本的应用程序提供了向后兼容性支持。

声明 schema 只是这个问题的一部分，你可能还需要以下几种机制。

- 注册

  新数据源及其 schema 的定义。

- 路由

  数据应该被发送到哪个主题、处理系统和存储系统。

- 采样

  这是对路由的扩展，可以删除部分数据，非常适用于暂存环境和测试。

- ❑ 访问控制

  谁可以访问数据流之外的持久化数据。

- ❑ 捕获元数据

  通过字段添加元数据。

在更高级的系统中，你还可以找到下面这些额外的功能。

- ❑ 转换逻辑

  在数据进入暂存区域之前，对数据进行自定义转换。

- ❑ 聚合和会话

  知道如何基于数据窗口执行操作的转换逻辑。

治理问题

随着需要收集、存储和分析的数据越来越多，对数据保护和隐私等问题的关注也在日益增长。因此，你需要制定计划以响应数据治理法规，并防止外部攻击、内部滥用等行为。你需要确保对所收集的数据有清晰的了解和归类。第 6 章将继续探讨这个话题。

5. 延迟和传递确认

与实时系统不同，数据管道在延迟和传递确认方面通常有很大的回旋余地。不过，在你确定系统期望时，数据管道的延迟和传递确认是非常重要的两个方面。接下来让我们定义这两个术语，并看看需要为此做些什么。

延迟

延迟是指从数据源发布信息开始，直到给定处理层或暂存系统能够访问到信息所花费的时间。为了更好地说明这一点，我们使用流处理应用程序作为示例。假设数据通过 Kafka 传入，然后由 Flink 或 Spark Streaming 消费。应用程序可能会根据处理结果向下游系统发送警报。我们可以将延迟量化为多个部分，如图 1-1 所示。

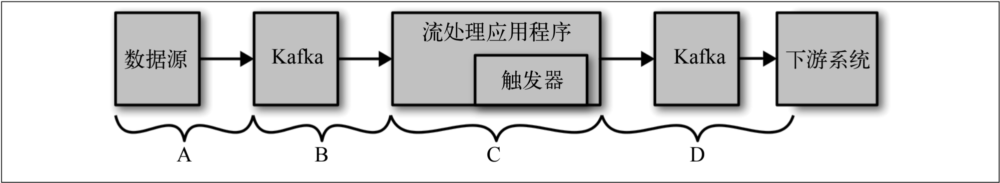


图 1-1：量化系统延迟

让我们来仔细看看每个部分。

- A：数据从源头到 Kafka 所花费的时间。这段时间就是我们所说的用于缓冲和网络传输的时间。但可能存在一种扇入式架构，其中包括负载均衡器、微服务或其他可能导致延迟增加的节点。

- B：经过 Kafka 所花费的时间。这取决于很多因素，例如 Kafka 的配置和消费者的配置。
- C：从处理引擎收到事件开始，直到它触发动作所需要的时间。有些处理引擎基于一定的时间间隔触发动作，比如 Spark Streaming；有些处理引擎则可以提供较低的延迟，比如 Flink。当然，这也受到配置和使用场景的影响。
- D：数据进入 Kafka，并从 Kafka 中读出。这个部分的延迟取决于生产者和消费者的缓冲及轮询配置。

传递确认

传递确认可以让数据源知道数据已经到达数据管道的哪个阶段，甚至可以让数据源知道数据是否已到达暂存区域。以下是在设计传递确认机制时的一些指导原则。

- 你真的需要传递确认机制吗？

  传递确认的好处是在发生故障时（例如网络问题或硬件故障），数据源可以重新发送数据。因为无法避免故障，所以传递确认机制可能适用于大多数情况。但如果这不是必需的，那么在实现数据管道时就可以节省很多时间，并降低系统的复杂性。因此，请务必确认真的需要传递确认机制。

- 如何进行传递确认？

  如果你确实需要进行传递确认，那么就需要在设计系统时考虑到它。这包括选择具备确认功能的软件解决方案，并将相关逻辑添加到数据管道的自定义代码中。因为提供传递确认机制会带来一定的复杂性，所以应该尽可能使用具备这项功能的现成解决方案。

6. 针对数据传递的风险管理

数据传递存在一定的风险。你希望能够为用户提供一个可以满足他们一切需求的系统，但在现实当中，总是会不可避免地出现一些错误，并且在某些时候，你的保证可能会失效。

有两种方法可用来应对这种风险。第一种方法是构建整洁的架构，将其分享给干系人，向他们征求有助于提升系统稳定性的意见。此外，适当的度量指标和日志有助于验证实现方案是否满足了需求。

第二种方法是提供一种机制，以便在发生传递确认丢失时能够通知用户和数据源，并提供做出调整或切换到备份系统的时间。

7. 目标数据的访问模式

数据管道的最后一个关注点是目标数据的访问模式。这里将重点介绍数据访问的类型以及在定义数据管道时需要考虑到的需求。

可以将问题分为两类：数据访问和数据保留。我们先来看看数据访问和最为重要的几种作业类型：

- 执行大型扫描和聚合的批处理作业；

- 执行大型扫描的流处理作业；
- 点数据请求，例如随机访问；
- 搜索式访问。

执行大型扫描的批处理作业

扫描大块数据的批处理作业是研究和分析数据的核心工作负载。这一类别包含 4 种典型的工作负载。为了更好地理解这个概念，我们以现实生活中的连锁超市作为例子。

❑ 分析 SQL

使用 SQL，并按照邮政编码和日期汇总已销售的商品。通常，这类报告需要每天运行，并提供给公司内部的领导层看。

❑ 会话化

使用 SQL 或 Apache Spark 之类的工具来**会话化**客户的购买习惯，例如，更好地预测他们的购物需求和模式，并提醒可能存在的风险（例如客户流失）。

❑ 模型训练

连锁超市的商品推荐系统可能会用机器学习模型分析顾客的购物习惯，并根据顾客的偏好提供商品建议。

❑ 场景预测评估

在连锁超市示例中，我们需要针对如何给商店里的货架补货制定一个策略。可以根据历史数据来判断采购模式是否有效。

本例的关键是需要很长一段时间内的数据，我们也希望能够为数据分析提供足够的处理能力，并为进一步行动提供参考。

执行大型扫描的流处理作业

批处理作业与流处理作业的区别主要体现在两个方面：作业执行时间和渐进式工作负载的概念。在时间方面，流处理作业通常被认为只需几毫秒到几分，而批处理作业通常需要几分到几小时，甚至是几天。渐进式概念的差异可能更明显，因为这表现在它们的输出结果不同。让我们来看看以下 4 种作业类型以及渐进式处理是如何影响它们的。

❑ 分析 SQL

通过采用流处理作业，能以更短的时间间隔（如秒或分）更新汇总报告，从而几近实时地看到变更，加快响应速度。此外，在理想情况下，流处理的成本并不会比批处理高太多，因为只处理增量数据，而不是重新处理旧数据。

❑ 会话化

与汇总报告一样，流处理作业中的会话化也以较小的时间间隔进行，从而产生更多实时的结果。在连锁超市示例中，我们使用会话化来分析顾客放进购物车的商品，以此来预

测他们可能会为晚餐购买哪些食物。有了这些信息，就可以为顾客推荐新上架的甜点。

- 模型训练

  并非所有的模型都适合进行实时训练。不过，可以实时运行训练好的模型，从而为业务决策提供实时的可参考结果。

- 场景预测评估

  如今，我们借助技术来实时地做出决策。我们还需要对决策进行实时的评估，这样才能知道是否需要对决策做出调整。随着决策制定越来越自动化，需要将其与实时评估相结合。

批处理需要大量的存储空间和大规模的计算，而流处理只需要足够的计算、存储空间和内存空间来处理给定时间窗口内的数据。



第 8 章将详细讨论流处理。

点数据请求

到目前为止，我们已经讨论了一些针对给定表、分区或流的访问模式。点数据请求是指在高并发的情况下快速获取特定的记录或数据点。

请想象一下连锁超市示例的以下这些场景：

- 在指定的商店库存中查找指定商品；
- 查询运送给指定商店的商品位置；
- 查询指定商店的商品供应链情况；
- 可以随时查询事件，并且能够向前和向后扫描，以便了解这些事件如何影响销售。

第 5 章将探讨与存储相关的内容，并介绍一些存储解决方案，这些解决方案非常适用于上述场景。在本章，我们只需要知道这种访问模式与及时获取实体的事件有关，并且能够实时地获取这些信息。

搜索式访问

最后一种访问模式，即最常见的搜索式数据访问。这种访问模式要求速度要快，同时还要为查询提供灵活性。在连锁超市示例中，你可能希望尽快找出库存中有哪些种类的蔬菜，或者来自某个受污染农场的所有商品。



第 5 章将详细地讨论针对这些场景的解决方案。

8. 针对访问模式的风险管理

之前的内容关注的是数据源和数据管道的创建，以便让系统用户能够使用收集到的数据。接下来探讨如何为数据用户提供数据。数据用户是指另一类完全不同的干系人。需要考虑的一个问题是，访问模式可能会因为有更大的用户群试图从数据中获取价值而更快地发生改变。这就是将访问模式分为 4 种类型的原因。如果你能够找到一些核心访问模式，并大规模使用它们，就可以将它们扩展到更多的场景中。

下面这些实践可以帮助你更好地管理访问模式方面存在的风险。

- 确保用户使用正确的访问模式。
- 确保存储系统具备可用性和弹性。
- 验证系统是否可以满足用户的需求。如果他们需要将你的数据复制到另一个系统来完成工作，那说明你的系统做得不够好。

1.2.2 数据管道和数据暂存团队的人员组成

我们已经介绍了与数据管道和数据暂存有关的主要考虑因素和风险，你可能还想了解这些团队需要哪些类型的工作角色。以下是一些可能存在于这些团队中的工作角色。

❑ 服务和支持工程师

服务和支持工程师的任务是与用户合作，包括源数据系统的干系人、访问系统数据的用户，等等。这些工程师的职责如下：

- 高效地使用系统；
- 为系统用户寻找好的应用场景；
- 与团队合作解决用户遇到的问题；
- 为用户提供建议；
- 帮助用户取得成功。

❑ 系统工程师或系统管理员

系统工程师或系统管理员痴迷于解决系统的运行时间、延迟、效率、故障和数据完整性等方面的问题。他们不需要太在意系统的实际使用情况，主要关注系统的可靠性和性能。

❑ 数据工程师

数据工程师非常了解系统中的数据和存储类型。他们的主要工作是确保用户能够正确使用系统提供的数据，以及确保大数据解决方案用在了正确的地方。他们是数据存储和数据处理方面的专家。

❑ 数据架构师

数据架构师是数据建模方面的专家，负责定义系统数据的结构。在某些情况下，团队中的数据工程师也可以承担这个角色。

1.3 数据的处理和分析

数据的处理和分析是指对数据管道和数据暂存项目中的数据进行转换和分析，以提取有用的价值。之前我们已经讨论过一些有关数据转换和价值创造的内容。不过，之前的讨论主要是关于如何为数据处理和分析应用程序准备数据。在本节中，我们将深入探讨这些转换和分析数据的应用场景。

1.3.1 主要考虑因素和风险管理

在评估这类应用场景时，我们主要关注以下事项：

- 定义要解决的问题；
- 通过处理和分析数据来解决问题。

简单地说，就是我们想要做什么、如何实现我们的目标、如何让这个过程可重复，以及如何量化它的价值。

1. 定义要解决的问题

你可能已经知道，很多问题如果可以得到解决，就能够为企业带来价值。但要确定需要解决哪些问题并非易事，特别是在有很多问题需要解决的情况下尤为困难。其中一些问题可能具有潜在的影响力，另一些可能很酷、很有趣，而有时候待解决的问题并不那么明显。

这里的关键不在于数据本身或者我们想要对数据做些什么，而在于我们想要从数据中获得什么样的价值、什么会对业务产生影响、什么可以作为行动的参考。要解决这些问题，我们需要与干系人——系统用户和客户——展开讨论。通常，我们面临的主要挑战是如何吸引这些干系人的注意，让他们愿意在这上面花时间。可惜的是，他们通常不知道自己真正需要什么。

在讨论过程中，可以向你自己或用户提出以下问题。

- 重要的跟踪指标有哪些？
- 影响顾客参与度的因素有哪些？
- 公司的产品是否存在差距？
- 是否存在痛点？

下一步是尝试设定进一步解决问题的目标，包括定义特定的指标、数字、可视化元素，等等。这样一来，就可以用它们评估解决问题的方法，并确定其他潜在问题与我们正在解决的问题相关以及如何相关。

让我们把这个想法应用到连锁超市示例中。假设我们的问题是货架上未能摆满合适的商品。可以创建如下的可视化元素来说明这一点。

- ❑ 商品的库存变化

  一张基于时间的图表，显示每种商品的平均库存量、最大库存量、最小库存量、前10%的库存量和后 10%的库存量。

- ❑ 缺货商品

  一张基于时间的图表，显示每种商品何时缺货。

借助上述两张图表，可以进一步创建有助于确定问题影响的可视化元素。

- ❑ 购买替代品模式

  在某种商品缺货时，顾客会选择其他商品吗？

- ❑ 货运供应链

  库存不足是交货延迟造成的吗？

- ❑ 地区或商店的差异

  各地区或各商店的库存水平是否存在差异？

- ❑ 顾客影响

  对于会正常购买某类商品的顾客，他们在商品有货和商品缺货时的支出是否存在变化？

我们主要是想提供有助于说明问题的背景，定义问题的影响范围，以及揭示问题的潜在影响，这样或许可以为制定决策提供足够的信息。这些信息还有助于确定这个问题与公司面临的其他问题相比孰轻孰重。

2. 针对定义问题的风险管理

在确定需要解决哪些问题时，有两个重要的考虑因素可以帮助你更好地管理风险，并使后续工作进展得更顺利。

- ❑ 了解许多人的观点

  确保你可以从多个来源了解到他人观点，而不只是一两个人。例如，你可以与高层管理人员或直接受问题影响的团队交流。但问题是，不同的人群与要解决的问题之间的距离要么太近，要么太远，所以他们无法从多个角度看待问题。获得额外的观点有助于在以后量化问题和解决问题。

- ❑ 建立信任

  你应该具有合作精神，不要试图从鸡蛋里挑骨头。你应该与干系人密切合作，获得他们的信任。

这一阶段的挑战在于确保能够正确地定义问题。一个有效的方法是频繁地公开讨论你要解决的问题。这种方法可能很常见，但它确实有助于避免将责任只归咎于组织的某些部分。有时候，将过多的注意力集中在一个特定问题上可能会将组织中的一部分人或部门置于不利位置。

要避免陷入这种办公室的勾心斗角，可以尝试找到其他部门中能够理解定义问题迭代过程的人，然后经常与这些人沟通，以减少令人不快的意外。

3. 实现和实施解决方案

现在，我们已经知道要解决哪些问题，并且有了解决方案，我们已经准备好实现这些解决方案了。将要解决的问题铭记在心，保持灵活性，并基于可能会影响解决方案的新信息做出调整。

在实现阶段有几个重要的考虑因素，就是要专注于构建健壮的解决方案，确保所构建的解决方案不是一次性的，并将解决方案付诸实施。

构建健壮的解决方案

构建只能解决单一问题的系统是一个常见的陷阱。更好的做法是构建可以解决多个问题的平台系统。能够快速而准确地获得结果固然是好的，但如果能够快速而准确地找到所有（或尽可能多）的答案会更好。

想要定义清晰的价值获取路径，可以尝试创建方块图来可视化，如图 1-2 所示。顶部是所有的数据源，当你从顶部移动到底部，就越来越接近价值和具有行动参考作用的结果。

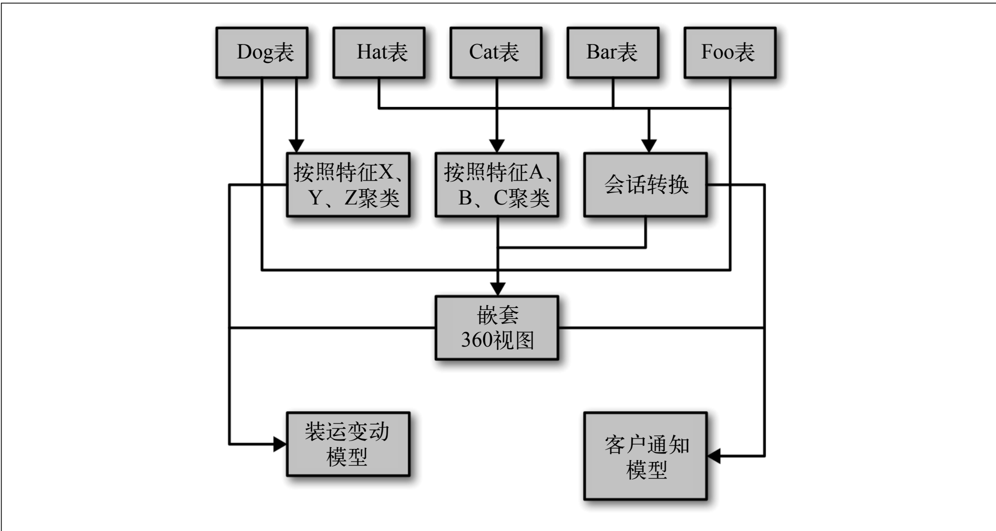


图 1-2：将系统分解为块

可以把图 1-2 中的方块想象成积木块。你可以利用像“嵌套 360 视图”这样的积木块来解决很多问题，不仅是当前的问题，还包括未来的需求问题。

构建过多的单一用途块存在一定的风险，因为你必须维护每一块。到了某个时候，你就需要将块合并，目的是简化价值获取路径。

如果想构建一个可以解决所有问题的系统，那么最终得到的系统可能什么问题也解决不了。先从小处开始，并确保你对业务需求有清晰的了解。

实施解决方案

让发现问题或解决问题的人来实施解决方案也是一个常见的陷阱，因为实施解决方案需要完全不同的技能。这两组人员之间需要建立良好的沟通渠道。另外，与寻找解决方案时相同，在实施解决方案时也需要投入巨大的精力。

1.3.2 数据处理和分析团队的人员组成

与数据管道和数据暂存团队不同，能够从数据中成功挖掘价值的团队更专注于发现问题、跨团队合作，以及寻找解决方案。这意味着团队需要不同的角色。

- 问题提出者

  问题提出者能够在公司中赢得不同部门的信任，并善于识别和量化问题。你可以把他们视为寻宝猎人。他们需要建立联盟、梳理日常业务，从而找到能够产生重大影响却待解决的问题。在团队中，他们可以是项目经理、产品经理或技术主管，也可以是开发人员和分析师。

- 架构师

  架构师能够成功解决问题的团队不仅能够正确地选择要解决的问题，而且会选择正确的顺序。这与搭积木如出一辙，我们总是选择只需要最少工作量的问题，而且这些问题包含了可重用的部分。

- 智囊团

  智囊团包括提出解决方案的数据科学家和分析师。

- 工程师

  工程师知道如何与上述各方合作以及如何开展工作和产品化。

- 方案沟通专家

  如前所述，在寻找解决方案的过程中可能会遭到他人的指责。此外，得不到支持的解决方案可能永远不会变成实际的产品。沟通专家负责传播解决方案，充分发挥它们的潜力。他们可能是项目经理、产品经理或技术主管。

1.4 应用程序开发

到目前为止，我们已经探讨了将数据传输到暂存区域的数据管道，以及专注于从数据中获取价值的项目。这两类项目更多的是关注数据收集和数据学习，而本节介绍的这类项目与部署使用数据向内部或外部用户提供服务的应用程序有关。基于数据驱动的网站就是一个

很好的例子。假设有一个支持大量用户的应用程序，它通过数据来驱动，同时还具备可扩展性、可靠性和可用性。

1.4.1 主要考虑因素和风险管理

这个用例包含以下这些有助于取得成功的关键考虑因素。

- 延迟和吞吐量

  执行一个操作需要多长时间，系统每秒可处理多少个操作？

- 局部状态和一致性

  如果系统在多个地区可用，那么它是如何进行复制的？它是孤岛式的、最终一致性的还是强一致性的？

- 系统可用性

  系统在发生故障和故障恢复方面有哪些特点？

1. 延迟和吞吐量

在构建应用程序时，可以先了解需要保存哪些数据以及如何与这些数据交互。与数据交互的操作包括插入、更新、多行事务处理，等等。

对于每种数据交互，都需要考虑很多潜在问题。

竞态条件

如果两个客户端同时更新同一行数据，谁会胜出？有几种办法可用于解决这个问题，请看以下例子。

- 最后一个胜出

  在这种情况下，谁输谁赢并不重要，它们只是随机地相互覆盖。

- 事务锁定

  这意味着在修改数据之前，需要在数据存储级别或服务器级别加锁。一个常见的例子是只有当某个条件成立时才能修改数据，例如，只有当 bar 列的值等于 100 时才能将 foo 列的值更新为 42。

异步操作与同步操作

在要求低延迟的系统中，进行实时的保存操作可能是不现实的。一些需要暂时保存在服务器或客户端的内存中，最后再异步持久化到最终的存储系统中。对于这种用例，需要考虑以下事项。

- 速度与事实

  在使用异步模型时，会有一个短暂的时间窗口，在这个窗口内可能会丢失数据。你需要评估数据丢失与延迟，哪个问题对于系统来说更严重。

性能一致性

性能是否具有一致性？你需要问自己以下这些问题。

- 如果数据的插入顺序变了会怎样？
- 性能会因为数据的扩展而发生变化吗？
- 性能会随着存储解决方案的变化而发生变化吗？
- 对于生成的数据和实际的数据，性能是否存在差异？
- 维护工作将如何影响性能？

2. 针对延迟的风险管理

要降低与性能相关的风险，最好的办法是及早进行测试和频繁地沟通。应该监控与性能相关的方方面面。你需要记录每一个连接的信息，无论是内部的还是外部的都要记录。另外，要对测试结果提出质疑，并让多个团队一起参与测试。

最后，确保在设计中使用接口（第 4 章将详细讨论），这样就可以无缝地替换接口实现。在构建出第一个解决方案后，你很可能会想到其他更好的策略。所以，给自己留一些空间，用于实现后面可能出现的变更，而不是重写整个系统。

局部状态

状态存在于多个地方，以下是分布式系统中存在状态的 4 个主要位置。

- ❑ 客户端

  用户正在使用的客户端接口。

- ❑ 服务器端

  客户端访问的服务器。

- ❑ 数据中心

  应用程序服务器所在的本地数据中心里的持久存储。

- ❑ 多数据中心

  跨数据中心的复制持久存储。

在制定这些场景的目标和需求时，需要考虑以下这些因素。

客户端

在客户端缓存数据，并允许在客户端修改数据，这样可以提升性能和可扩展性。但是，客户端状态也存在一些潜在的问题。

- ❑ 临时性

  客户端随时都可能发生故障，导致数据在到达服务器之前丢失。

- ❑ 信任

  如果客户端信任对你来说很重要，那么在不受信任的主机上运行的客户端（包括不受信任的用户）就会带来一些问题。

服务器端

虽然服务器端不存在信任问题，但仍然可能会发生临时的数据丢失。服务器端的另一个潜在问题是用户分区。客户端按照用户分区，而服务器端按照用户组分区。在某些情况下，用户分区可能是基于某些明确的策略。例如，对于游戏应用程序来说，就需要在同一台服务器上维护用户状态。但在很多情况下，用户状态可以随意放置。例如，在 Web 应用程序中，可以基于负载均衡器放置用户状态。

在需要多服务器状态管理的场景中，可能需要考虑数据中心持久存储。

数据中心

在这一层，你可以看到一些用于持久化状态的常用技术，例如 NoSQL 系统、关系数据库和分布式缓存。这一层的目标是存储当前区域以及当前区域内的服务器和客户端所需的所有状态。

数据中心里的数据很可能会被复制，以避免受一个或多个节点故障的影响。但这并不意味着这种保护措施是完美的，因为整个区域也有可能发生停机事故。如果你看重这个问题，那么可能需要采用多数据中心。

多数据中心

有几种模型可用于多数据中心场景，具体使用哪一种取决于用例需求。接下来介绍一些可用的模型。

- ❑ 灾备恢复复制

  对于这种情况（例如在整个数据中心变得不可用时），我们使用多数据中心配置来防止数据丢失。这通常是一个异步或批处理的过程，数据最终会在数据中心之间保持一致。我们不要求局部的完全一致性，但要朝着这个方向努力。虽然这样仍然无法完全避免数据丢失，但确实可以避免大部分的数据丢失。

  只使用复制时，需要考虑这样一个问题：如果多个区域同时修改了数据，会发生什么？因为没有全局一致的状态，所以也就没有单一的事实来源。

- ❑ 锁定

  全局锁定是一种在发生跨区域数据变更的情况下仍然能够保持一致性的解决方案。它的主要思想是全局锁定某个资源，在获得锁之前，没有哪个区域可以修改这个资源。有多种方式可以实现这种锁定机制。

  - 客户端锁定

    每个客户端都需要获得锁才能修改给定的记录。

- 数据中心锁定

  所有的数据变更都必须先通过指定的数据中心。这比客户端锁定使用更少的锁，但会给位于当前区域之外的客户端造成更高的网络延迟。

- 记录级别的锁定

  这其实是一种仲裁架构。我们有奇数个状态存储，只要多数区域同意当前的状态变更，这个状态变更就会被接受。

需要注意的是，数据中心里的数据通常仅限于当前区域内的用户访问。如果你的应用程序涉及不同区域客户端之间的交互，那么就需要考虑不同区域如何共享数据。

3. 针对局部状态的风险管理

在规划项目时，最好能够尽早制定有关状态管理的策略。项目的要求和目标是什么？有关状态的决策将如何影响用户？在项目启动之后就很难再改变这些策略。

做出决定后，你需要将这些决定完整地记录下来，并列出可能对用户造成的影响。文档需要采用所有用户都可以访问并且能够清楚传达决策影响的格式。

4. 可用性

系统可用性是一个很关键的因素，但同时也颇具挑战性。影响系统正常运行的因素有很多，包括如下这些因素。

- 人为错误

  人都会犯错误，比如糟糕的配置变更、错误的代码部署，等等。

- 升级

  某些升级需要重启系统。即使是在滚动重启的情况下，系统的某些部分在一段时间内也是不可用的。

- 故障

  无论是在本地还是在云端运行系统，硬件故障都不可避免。

- 攻击

  在规划系统时，也需要把恶意攻击考虑在内，它们有可能会影响系统的可用性。

你需要定义故障和可以满足给定服务等级协定的恢复方案。以下是故障点和恢复计划的一些示例。

- 服务器故障转移

  如果当前服务器出现故障，要能够启用另一台服务器。如果状态位于数据中心里，那么单台服务器发生故障几乎不会造成任何影响。

- ❑ 非复制缓存故障转移

  有些设计方案将状态放在缓存中，当缓存因为故障而被销毁时，状态数据可能被持久化到其他存储中。不过，从存储中恢复缓存需要花费一点时间。

- ❑ 最终一致复制的数据中心故障转移

  最终一致复制是在广域网上复制数据的常用方法。如果发生故障，可以将请求切换到不同的数据中心。但多数据中心故障转移存在两个问题。首先，在发生故障时，发送出去的数据很有可能会丢失；这个时间窗口可能很小，具体取决于实际的吞吐量和延迟。其次，应该如何处理写入操作，例如，是选择将写入操作重定向到负责管理写入操作的首领中心，还是重定向到任意一个数据中心。

5. 针对可用性的风险管理

要解决可能影响系统可用性的问题，一个比较好的策略是有意识地定期向系统中引入故障，例如，使用由 Netflix 开发的 Chaos Monkey 故障注入工具。

故障测试的结果应该被用来定义故障影响、恢复计划以及可以在未来缩小故障影响范围的措施。将这些作为系统的一部分一起发布，使用户对系统的预期更实际。

此外，如果做得好，还可以形成一种文化。在这种文化之下，人们不仅将故障作为高优先级事项来考虑，还会积极采取措施缩小故障的影响范围。

1.4.2 应用程序开发团队的人员组成

与其他两个用例不同，应用程序开发关注的是用户影响、一致性、行为和数据移动效率。应用程序开发团队的人员配备可能有一部分与数据管道和数据暂存团队的相同，但仍然存在一些明显的差异。具体地说，应用程序开发团队可能包含以下角色。

- ❑ 网站可靠性工程师

  网站可靠性工程师致力于保证部署在生产环境中的应用程序具备可靠性和可扩展性。他们是成功部署应用程序的关键。

- ❑ 数据库工程师

  数据库工程师不是传统的数据库开发人员和架构师，而是对现代分布式数据存储和处理系统有深入了解的人。他们需要确保数据库能够快速执行读取、写入和事务处理等操作，以此来满足应用程序的需求。

1.5 小结

本章很长，讨论了很多话题。重点在于，我们在实施项目之前要对项目有很好的了解，并经过了周密的计划。为此，本章将数据项目分为 3 种最常见的类别。

- 数据管道和数据暂存

  这类项目将源数据引入到系统中，并为进一步的处理做准备。这类项目将为其他类型的项目奠定基础，因此在规划时要十分小心。

- 数据的处理和分析

  在有了可用的数据后，这类项目通过处理和分析从数据中获取具有行动参考价值的见解。它们可能是分析师用于探索数据的临时项目，也可能是为业务用户提供报告和仪表盘的完整项目。

- 应用程序开发

  这类项目面向用户开发应用程序，为内部用户或外部用户提供服务和价值。它们通常依赖于前面两类项目的成功实施和部署。

我们针对每类项目讨论了有助于推动项目规划和开发的注意事项，并将考虑因素分为 3 类。

- 主要考虑因素

  在规划每类项目时应该考虑的特殊注意事项。

- 风险

  应该纳入到项目规划中的潜在风险，以及可以降低这些风险的信息。

- 项目团队

  在组建团队以交付项目时需要考虑的角色类型。

本章所提供的指南基于我们从不同公司的多个项目中总结出来的经验，它们应该可以帮助你成功地实施数据项目。接下来将针对其中一些主题进行更详细的介绍。

第2章
评估和选择数据管理解决方案

你可能已经意识到，在开始一个新项目时，技术选型有多么重要。选择正确的解决方案是一个复杂的过程，它会对组织产生长远的影响，并直接影响数据项目能否取得成功。是重用可信赖的解决方案，还是尝试新方案，或是步行业领头羊的后尘？在选择解决方案时，我们必须面对这些艰难的决策。

在决策过程中，你可能会看到市场炒作、来自供应商的夸大承诺、能够提供不同解决方案的分析师以及拥有忠实用户基础的工具。其中一些解决方案可能适合你的项目，有些则可能不太适合。后者只会浪费时间和金钱，让你感到挫败。

本章的目的是帮助你在技术选型过程中做出最好的选择。首先，讨论开源项目的一些常见的生命周期，这些知识有助于了解开源项目的健康状况、项目处在生命周期中的位置，以及抛开炒作（不管有没有）看看项目是否适合系统。然后，讨论用于评估项目的一些指标，例如性能测试或基准测试的结果。最后，深入探讨一些项目技术选型模式。

你很可能会问：为什么本章要关注开源项目？可以采用商业解决方案吗？关注开源项目的主要原因是，很多较新的大数据平台都是开源项目，而且很多现代专有解决方案也是基于开源软件或利用了开源软件，在某些情况下甚至会复制开源解决方案的功能。当然也有例外——Teradata 是一个闭源的商业数据管理解决方案，它可以存储和处理海量数据。在云计算领域，亚马逊 Web 服务（AWS）提供了专有产品，例如简单存储服务（Simple Storage Service，即 Amazon S3）和 Kinesis，它们主要用于存储和处理大数据。然而，这些专有软件的开发通常遵循与开源项目类似的路径，只是被隐藏在公众视野之外。本章将介绍开源产品和闭源产品之间的差异。

虽然我们主要关注开源项目，但并不是说第三方商业解决方案就一定不适合你的数据项目。大多数企业软件供应商支持主要的大数据平台，并且通常会提供可以促进应用程序开发的解决方案，特别是当你已经在一个组织的其他方面使用了这些解决方案。另外需要注意的是，很多与开源项目相关的决策点也适用于商业产品。

另外，很多软件供应商采用混合模型，即将专有解决方案和开源解决方案相结合。回到 Teradata 的例子，虽然它的旗舰产品是专有的，但它同时也提供了开源产品。例如，它支持 Apache Presto 和其他开源系统。另一个例子是 AWS，它除了提供专有服务，还提供基于开源软件（如 Apache Hadoop）构建的服务。

在阅读时还需要注意，本章提到的大多数考虑因素更适用于在本地或云端部署的软件。这与云服务供应商或托管服务供应商提供的托管服务恰好相反。虽然本章以及其他各章讨论的很多考虑因素也适用于托管服务，但你的选择在很大程度上将受到供应商提供的服务的限制。

2.1 开源项目的阶段

在参与多个开源项目之后，我们发现开源项目往往会经历一些共同的阶段。并非每个项目都会经历所有这些阶段，而且其他人可能会总结出不同的阶段，但在本章中，我们将这些阶段分为**孵化阶段**、**发布阶段**、**“治愈癌症”阶段**、**打破承诺阶段**、**强化阶段**、**企业阶段**和**终结阶段**。本节将详细解释每一个阶段，以帮助你确定自己的项目正处在其生命周期的哪个阶段，从而对项目做出评估。

在了解这些项目生命周期阶段的同时，你可能会注意到它们与 **Gartner 炒作周期**有一些相似之处。Gartner 炒作周期是由 Gartner 公司创造的一种方法，用于跟踪技术的成熟度。以下是 Gartner 炒作周期的各个阶段，以及它们与本章所述项目阶段之间的关系。

- 技术的触发
  孵化阶段和发布阶段。
- 膨胀期望的高峰
  “治愈癌症”阶段。
- 幻灭的低谷
  打破承诺阶段。
- 启蒙的斜坡
  强化阶段。
- 生产力的高原
  企业阶段和终结阶段。

2.1.1 孵化阶段

很多成功的开源项目是从内部孵化项目开始的，并且通常由母公司或大学提供赞助。母公司通常会提供初始资金，同时也会提出项目要解决的问题。这方面有很多很好的例子：Apache Kafka 最初是在 LinkedIn 开发的，用于解决公司在数据集成方面所面临的问题；Apache Hive 来自 Facebook，其目的是方便用户访问数据；Apache Spark 来自加州大学伯克利分校；Apache Impala 的前身是 Cloudera Impala。

很多项目在经历了内部孵化后会进入外部孵化阶段，不过，内部孵化阶段既有好处，也有一些不足。内部孵化阶段设定了项目的发展方向，建立了一致的架构，为开发人员提供了可遵循的开发风格和用于构建可靠代码库和文档的时间，等等。这也有助于阐释项目能够提供的价值以及可以解决的技术问题——即使是对于开源软件来说，营销也至关重要。

既然孵化阶段对于大多数人来说是不可见的，那为什么要在讨论开源项目时提到它呢？因为这个阶段对项目的成功至关重要——从长远来看，项目在顺利度过孵化阶段后更有可能获得成功。来自母公司的支持可能是项目头几年的推动力。

2.1.2 发布阶段

这是项目开源之后的阶段。对于处在这一阶段的项目，需要了解如下重要的方面。

- 项目获得了多少支持和动力？

  是否有与项目发布相关的文章？项目是在行业大会上出现过，还是只有少数人知道？此外，赞助实体在支持项目方面是否具有良好的声誉和记录？同样，项目的核心开发人员是否具有良好的声誉？

- 项目的定位和愿景是什么？

  项目是否清楚地说明了它试图解决的技术问题？这一点很重要，因为它将在项目的发展方向以及技术能否被采用等方面起到重要作用。

- 项目是演化式的还是革命性的？

  当一个项目刚刚出现时，需要将其与声称可以解决相同或类似问题的项目进行比较。新项目所做的是不同于已有解决方案的重大改进，还是渐进式的改善？之所以要问这个问题，是因为切换技术需要耗费一定的资源。通常来说，如果只是渐进式的改善，那么就不值得切换技术。我们需要问自己，市场上现有的项目是否也能解决新项目要解决的问题，它们之间的差距是否大到只能由新项目来填补，以及新项目能否带来已有项目无法匹敌的好处。

2.1.3 “治愈癌症”阶段

无论好坏，大多数项目会经历一个可以被称为“治愈癌症”的阶段。在这个阶段，新项目进行积极的自我推销，强调（有时过度夸大）项目能够提供的好处。如果看到以下这些内

容，说明项目正处在这个阶段。

- 项目承诺为每个问题提供简单的解决方案。
- 很少提及项目的局限性或问题。
- 技术工作者对项目很感兴趣，需求量也很大，却不清楚项目将如何为他们提供价值。

实际上，这个阶段对于开源项目来说很重要。如果没有这种炒作，项目很可能无法吸引到进入下一阶段所需的提交者和贡献者，也无法获得进入下一阶段所需的采用量。

作为决策者，应该如何看待处于这个阶段的项目？一种选择是谨慎、乐观地对待项目。对项目进行谨慎而全面的评估，看看它能否为要解决的问题提供解决方案，又或者它只是一种炒作。2.4 节提供的指南为评估处于这一阶段的项目提供了参考。

另一种选择是先等待一段时间。这通常是更好的选择，因为随着项目或技术日益成熟，实现价值的时间会不断缩短。出现这种情况的原因有很多：更少的错误、更多的示例和文档、更多训练有素的技术人员、更少的 FUD（恐惧、不确定性和怀疑）。

除非公司的业务非常前沿，否则如果项目处在这个阶段，在开始评估项目之前，请先等待一段时间（至少 6 到 12 个月）。等到其他人先碰完该碰的壁，你就会发现，成功之路会更加顺畅。此外，等待只会让你更快地抵达目的地，而且可以获得比早期采用者更好的架构，因为有了更强大的知识体系作为基础。

2.1.4 打破承诺阶段

经历过炒作阶段之后，大多数项目会变得更实际。一般来说，如果选择的开源项目没有打破过某种承诺，那说明还没有充分使用它。当发生以下情况时，就说明“梦想不再照进现实”。

❑ 向新用户开放

当项目开始向开发人员和有经验的用户之外的更大用户群发布时，项目代码库和文档的相关问题及局限性就开始显现出来。

❑ 大规模运行

无论在系统的设计和架构方面做出多大的努力，都很难对项目将在现实世界中面临的工作负载类型进行全面的建模。受制于生产环境工作负载的项目很容易出现伸缩性问题。在最坏的情况下，系统可能需要进行重写或重构才能满足这些预期之外的规模需求。

❑ 安全

对于很多开源项目来说，安全问题的优先级通常低于功能。安全问题不一定是导致项目失败的原因，这只是一个务实的选择，主要是为了能够尽可能方便地解决问题。如果项目发展成为企业解决方案，那么这就会成为一个问题，而且在将安全纳入到系统中时需要做出更大的努力。

- 审计和维护

  对于企业解决方案来说，审计和维护很重要，但与安全问题一样，它们通常不是开源项目初始开发阶段优先考虑的方面。很多针对企业市场的开源项目在这方面做得越来越好，但这个领域仍然存在需要填补的空白。

- 集成

  解决方案可能都会存在一些局限性，这些局限性导致在将解决方案与现有系统和架构集成时颇具挑战性。

打破承诺阶段是大多数开源项目的成败点，它决定了项目能否存活下来并解决所有出现的问题。

请注意，这个阶段是任何一个成功项目都必须经历的阶段。几乎可以肯定的是，你所评估的任何解决方案都将经历这个阶段，不应该将处于这个阶段的项目视为不合格的候选项目。如果项目到达这个阶段，说明人们正在使用这个项目查找问题、了解项目的局限性，等等。这对于项目进入强化阶段来说是必要的。

最后，这个阶段为采用者提供了在社区中发挥重要作用的机会，并随着项目的进展为后续用户提供帮助。

2.1.5　强化阶段

在强化阶段，早期开发变成了增量式开发。这个阶段并不那么令人兴奋，但如果项目要在企业中得到广泛采用，就需要经历这个阶段。

在这个阶段，项目的重点将放在审计、安全性和弹性方面。所幸正如在很多项目中看到的那样，通常可以在后续阶段添加这些特性，而无须重新设计解决方案。让用户最头疼的问题是添加这些特性的速度——这些特性都很复杂，需要在项目中仔细加以监督。

有趣的是，项目的原始开发团队可能并不是完成这些强化工作的团队。随着项目日益成熟，项目发起人和早期开发人员可能会转向其他的角色和项目。只要项目建立了稳固的社区和健康的生态系统，就可以稳步过渡。事实上，项目可以脱离初始开发人员正常运作是取得成功的好兆头。

在很多情况下，你会发现处于这个阶段的项目依赖于两组提交者：一组专注于强化，另一组则专注于创新。后者仍然在尝试突破软件的功能边界，扩大软件的应用范围，而前者努力让项目成为企业级软件。

随着时间的推移，强化组很可能会有所增长，创新组则会收缩。创新组将转向更自由和开发速度更快的子项目或新项目，因为这就是创新组的天性。

请记住，这两个群体都很重要。要保持它们之间的平衡，就需要密切关注以下因素。

- 平衡

  如果没有持续的创新，项目将会比预期更早过时。如果没有进行务实的开发，软件将永远无法在生产环境中运行。

- 技术债务

  项目进入强化阶段所需的时间越长，或者项目专注于创新的时间越长，技术债务就会越多。到了某个时刻，就不值得花成本去解决技术债务了，项目将走向死亡。

- 应该信任谁并与他们合作?

  在这个阶段，项目的某些部分是稳定的，其他部分则不稳定。应该听取创新者的意见还是实用主义者的意见？虽然双方都认为项目具有巨大潜力，但他们对项目适用的场景有不同的看法。你需要同时听取他们的意见。但如果打算部署产品，则要着重关注实用主义者的意见。可以听一听创新者的意见，但那只是为了了解可能性和未来的发展方向。

2.1.6　企业阶段

经历前一个阶段需要克服巨大的挑战，所以很多项目未能到达企业阶段。一款产品是否已经到达这个阶段很容易就能看出来：它可能已经被成功部署到很多公司的生产环境中。在几乎不参与这些项目开发的组织中部署这些项目是非常重要的，因为这可以告诉我们如下这些事情。

- 软件得到了支持和维护

  如果部署软件的公司不是软件项目背后的开发驱动力，那么就是其他公司或力量在为软件提供支持。

- 软件是稳定的

  项目稳定的一个很重要的迹象是，它被部署到金融服务公司的生产环境中。这是因为，对于这些公司来说，采用不稳定的技术可能会造成巨大的损失。

- 实用主义者多于创新者

  通常到了这个阶段，实用主义者的数量会超过创新者的数量。这表明对稳定性的需求在增长，因为现有产品已经可以满足市场的需求，对新功能的需求在减少。

很多数据管理系统已经到达了企业阶段，例如 Apache Hadoop、Apache Cassandra、Apache Kafka，等等。

2.1.7　终结阶段

开源项目的最后阶段有点令人沮丧。虽然项目被宣告死亡或完全关闭的情况极为罕见，但它们的发展速度会放缓，直至停止创新。

项目之所以会进入这一阶段，一个主要原因是它们不再是特定领域的市场领导者。这可能是因为项目创新不足，项目的底层硬件假设发生了变化，或者代码库和项目的繁文缛节阻碍了创新。

如果项目到达了这个阶段，并不一定意味着要开始寻找其他的解决方案——要知道，人们现在还在使用大型计算机！关键是要了解项目是如何进入这个阶段的，并在市场上寻找能够给未来应用场景带来价值或可以重新实现现有解决方案的项目。

另一个重要的考虑因素是这些解决方案对架构的重要性。一方面，如果项目处于架构的核心，就需要仔细考虑在什么时候弃用它。另一方面，如果它起到的作用较小，那么过渡到新的解决方案可能会更有意义。

2.2　开源项目的常见生命周期

在定义开源项目的主要阶段后，是时候讨论一些例外情况了。并非所有的开源项目都会遵循之前描述的整个生命周期。关键是要理解为什么有些项目会遵循不同的路径，这会影响对项目当前状态的理解。

在介绍例外情况之前，先来回顾一下开源项目的默认路径，如图 2-1 所示。

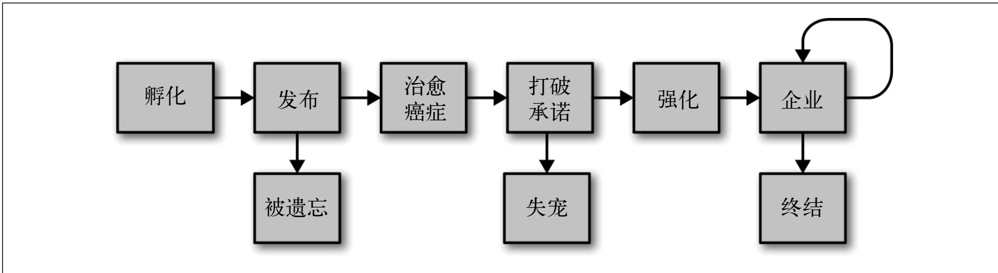


图 2-1：默认的开源项目生命周期

仔细看一下生命周期中的每个阶段。

- 孵化阶段

  作为组织的内部项目，由组织提供支持。

- 发布阶段

  成为开源项目。

- 被遗忘阶段

  可悲的是，虽然大多数开源项目在刚开始时有很好的意图，但从未得到广泛采用，很快就被遗忘了。

- 治愈癌症阶段

  通过炒作（很可能是过度炒作）让项目获得外部采用，引起更多人的兴趣，从而吸引更多的贡献者。

- 打破承诺阶段

  最初没有参与项目的人开始使用项目，项目的缺陷开始涌现。

- 失宠阶段

  成功的项目能够将打破承诺阶段作为跳板，将自己变成更可靠的项目。而那些无法克服问题的项目终将被放弃和遗忘。

- 强化阶段

  解决项目缺陷的漫长过程，如审计、安全和故障转移。

- 企业阶段

  项目最富有成效的阶段。在这个阶段，项目的大部分炒作已经消停，大多数人已经清楚项目可以做些什么以及不适合用来做什么。

- 终结阶段

  开源项目生命周期的最后阶段。在这个阶段，贡献开始放缓，人们的兴趣转向了其他新技术。

在定义开源项目的默认生命周期后，让我们来看看应该注意的其他生命周期。

2.2.1　使产品起死回生

近年来，有些公司已经从专有解决方案转向了开源解决方案，而且不仅仅是大数据领域如此。开源软件为公司提供了诸多优势，例如更低的成本和更少的供应商锁定。这种情况导致的结果是：一些产品的市场份额被开源项目夺走。要重新夺回市场份额，一种方法是重新启动产品项目，有时候甚至要改个名字，并作为开源项目开放出来。

但实际上，这样做很少会起作用，原因如下。

- 产品的市场份额已经枯竭

  仍然可以为公司提供价值的产品将继续占有市场份额，否则，公司会寻找更强有力的解决方案。

- 闭源与开源的开发方式截然不同

  商业软件和开源软件之间存在编码实践和文化方面的差异。在从一个模型转向另一个模型时，这些差异可能是难以弥合的。

- 不愿放手

  通常，在开源一款产品时，产品背后的公司不愿意放弃控制权，而放弃控制权是吸引外部开发人员所必需的。在不放弃控制权的情况下开放项目的方法是只开源产品的一部分代码，或者将代码托管在公共的代码仓库中（如 GitHub），而不是捐赠给像 Apache 软件基金会这样的组织。

- 老化的解决方案

  新想法让开源世界变得令人无比兴奋，而旧产品不太可能如此。

这些问题将导致开源项目出现不同的生命周期。在这种情况下，生命周期看起来更像图 2-2 所示的那样。

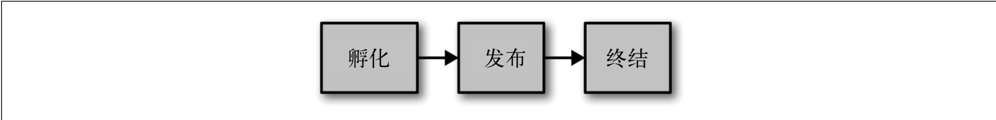


图 2-2：使产品起死回生的生命周期

2.2.2 追随者

虽然企业通常使用开源软件来降低成本，但仍然可以从开源市场中赚到钱。因此，很多项目试图从领导者那里争夺市场份额——我们将这些项目称为“追随者”，不过将它们称为“竞争对手”会更体面一些。追随者通常宣称自己比领导者更好，因为它解决了领导者遗漏的问题。追随者项目的生命周期看起来更像图 2-3 所示的那样。

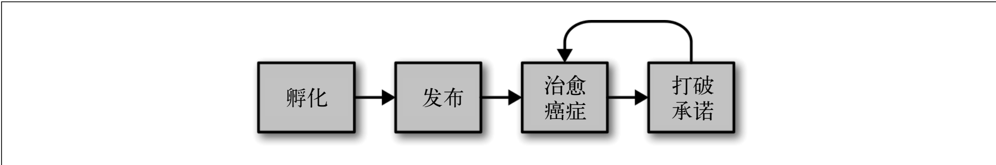


图 2-3：追随者项目的生命周期

追随者项目可能会长期处于“治愈癌症”阶段。这是因为它很难获得像领导者项目那样的采用率，所以还未能到达打破承诺阶段。

应该关注追随者项目吗？答案是视情况而定。这些项目可能会获得一些关注，但要注意它们是否到达了打破承诺阶段，并开始冲破这个阶段。对这类项目要保持谨慎，因为它们当中有很多往往很短命。此外，一些追随者项目只是从领导者项目那里借鉴想法和实现方法，这类项目很可能在长时间内无法超越领导者项目。

尽管以上描述令追随者项目看起来不太乐观，但一定会有取得成功的追随者项目，它们甚

至可以凭借实力超越自己并成为领导者项目。Apache Flink 就是一个很好的例子。它最初似乎只是试图填补与 Spark 相同的利基市场，因为它提供了类似的 API。但是，随着时间的推移，Flink 为很多应用场景提供了有用的工具，特别是在流处理方面。作为批处理引擎，Spark 可能比 Flink 更强大，但 Flink 提供的流处理应用程序在功能和性能上都具有优势，因此可以考虑将它作为 Spark Streaming 的替代方案。

此外，在某些情况下，领导者在追随者面前会显得黯然失色。在选择项目时，人们会考虑选择追随者项目，而不是领导者项目。Apache Storm 就是这方面的一个例子。长期以来，它被视为流处理应用程序的强有力竞争者。但近年来，其使用量在减少，人们转向了更新的解决方案，如 Spark Streaming。Spark Streaming 和 Flink 这些竞争对手都在蓬勃发展，而 Storm 这样的领导者似乎正在走向终结阶段。

2.3 评估基准测试

无论项目处于生命周期的哪个阶段，都会不可避免地出现有关性能比较或基准测试的噪声。基准测试有一定的参考价值，但我们需要抱着怀疑的态度来看待它们。

一方面，基准测试可以告诉我们哪些技术在市场中具有竞争力，以及如何得出这些结果。另一方面，有很多因素会影响基准测试的有效性。

- 因知识或方法不完备而导致的错误

  执行基准测试的人可能对软件没有充分的了解，无法公平地对待所有测试。虽然测试人员希望进行公平的测试，但由于误解以及存在缺陷的方法，导致出现无效的结果。例如，测试人员因为经验和知识不足，无法对测试涉及的所有工具提供相同级别的调优。另外一个例子是评估竞争产品。

- 无意识的偏见

  无意识的偏见会导致糟糕的测试方法、对竞争产品做出不恰当的调优，等等。这些都会导致结果无效。

- 动机偏见

  在某些情况下，偏见导致的无效测试可能是有意而为的，还可能为了展示特定工具的优势而有意进行测试。当执行测试的组织与测试结果有利益关系时，很容易出现这种情况。

- 不公平的比较

  在比较两种工具的基准测试中，有时候只采用了有利于其中一方的用例或工作负载，这也是一种很常见的情况。

尽管存在上述问题，基准测试仍然可以在以下这些方面为我们带来价值。

- ❑ 了解市场

  基准测试可以告诉我们哪些项目具有可比性，从而知道市场上有哪些竞争对手。

- ❑ 找出市场领导者

  如果有多个基准测试都在与某个工具作比较，那么这个工具很可能就是市场领导者。

- ❑ 测试脚本

  良好的基准测试会将测试使用的代码、配置和测试数据共享出来，其他人可以使用它们来验证测试结果。如果正在评估的基准测试没有发布这些内容，那么它的结果就非常可疑。出于可重复性和可验证性方面的考虑，基准测试应该遵循某些开放标准。

最后，需要基于数据针对具体应用场景进行内部基准测试。有些应用场景很独特，公开的基准测试可能不适用。通常，可以与供应商或外部专家合作，为要测试的工具提供配置、调优等方面的帮助。

2.4 技术选型的考虑因素

我们已经讨论了开源项目的生命周期和基准测试的真实情况，那么如何将它们应用在技术选型过程中呢？在回答这个问题之前，先来看看一些会推动和影响决策的因素。

- ❑ 业务需求

  要实现的特定用例和需求。

- ❑ 内部需求

  团队想解决新的问题和学习新的技术。

- ❑ 渴望保持前沿

  想被视为公司的先锋。

- ❑ 风险承受能力

  想避免失败和当众尴尬。

- ❑ 压力承受能力

  希望系统能够顺利地从开发环境进入生产环境，没有人愿意因为未知的问题而在凌晨一点钟接到求助电话。

- ❑ 技能差距

  根据员工现有的水平，选择他们能够掌握的一组技术。

在考虑上述因素的同时，可以使用业务需求来定义一个范围，从而确定哪些工具与选型过程相关。这个过程需要考虑到功能、成本和可扩展性等方面。以下将提供一个指南，帮助你在选型过程中进一步评估解决方案。

2.4.1 了解构建块

技术发展得很快，不过新系统通常建立在已有的架构和概念之上。以 Apache Kafka 为例，虽然它可以比以往的系统提供更高的数据吞吐量，但它的核心实际上是一个分布式日志。它借鉴了传统的发布 – 订阅消息传递系统的概念。

这个例子有点过于简单，但关键在于，在理解一项复杂的技术时，首先要将它分解，然后将各个部分与我们已经了解的系统进行比较。在获得实践经验之前，这个过程有助于了解系统的潜在局限性和优势。即使是全新的系统，这些基本原理仍然适用。

评估大数据解决方案的主要组成部分

为了提供更多细节，下面列出了一些示例，用于说明在评估大数据解决方案时需要注意的事项，其中包括了一些主要的组件。这并不是一份详尽的清单，只是为了展示在评估技术解决方案时需要考虑的事项。

存储系统组件

- 数据与数据消费者的相对位置

  数据与数据消费者之间有怎样的位置关系？举个例子，直到最近，数据局部性才成为 Hadoop 分布式文件系统（HDFS）架构的核心。在当今的云计算和动态计算资源领域，转向远程存储是一种趋势。这两种架构方法都有各自的优点，因此需要根据应用场景和需求对此做出评估。

- 压缩格式和压缩速率

  并非所有的系统都采用了相同的压缩方式。它们可能使用了相似的压缩编解码器，但预压缩数据的格式会对压缩产生巨大影响。

- 数据结构

  数据的布局方式会对访问模式和存储成本产生巨大的影响。第 5 章将讨论与存储系统相关的问题。

- 分区、复制和故障恢复

  用于分布式系统分区、复制和故障恢复的模型只有几种。对解决方案进行分类，并根据应用场景来比较解决方案的优点和缺点。

- API 和接口

  API 和接口是进入存储系统的主要窗口。请确保它们对系统是友好的，否则，你可能会拥有一个很出色但对你毫无用处的存储系统。

处理引擎

- 资源分配

  资源分配将极大地影响系统的可扩展性以及同时支持多个用户的能力。

- 重排和优化

  分布式系统的主要价值之一是能够优化作业并根据不同的目的来移动数据。需要对它进行分类，并与其他工具进行比较。

- 用例分类

  不同的执行引擎侧重于针对不同的用例进行优化。

- API 和接口

  同样，系统必须易于使用，并且可以使用已有的工具进行访问。

2.4.2 寻求建议

作为技术决策者，你一定知道这个世界上并不存在灵丹妙药。不过，即使是最有经验的技术专家，也可以从外部指导中有所收获。与顾问或分析师合作会有帮助——他们一直在关注技术发展的起起落落，能够提供更清晰的技术方向。

2.4.3 从分析师那里获得见解

Gartner 和 Forrester Research 等行业分析公司是评估和选择解决方案的另一个潜在指导来源。可以参考这些公司发布的报告，或者与这些公司的分析师会面，从而得知他们的分析结果。一般来说，这些分析师通过与供应商和客户召开会议来获得分析结果，不过在某些情况下，他们也会自己去执行测试。这个过程能为分析师提供广阔的市场视野，包括市场的现状、哪些工具正在引领市场，以及企业是如何使用这些工具的。

这个过程可以产生有价值的见解，并让分析师成为宝贵的资源。但在很多情况下，分析师的数据主要来自与供应商和客户的对话，而不是来自实际使用产品的经验。所以，这可能导致偏见。例如，供应商对自己的产品有强烈的偏见，客户也无法客观地评估自己的决策，而倾向于努力捍卫自己的技术选择。

分析师会受到有意识和无意识的偏见的影响。这可能是因为供应商提供了错误信息、与供应商或客户之间的关系，以及掌握的知识不充分或时间不足。分析师提供的一些据称客观的报告实际上也有特定供应商的参与，而这种情况并不罕见。

因此，可以将分析师视为有价值的资源，但请将他们的意见视为综合评估过程中的另一个可变因素。

2.4.4 研究市场趋势

到目前为止，我们已经讨论了一些用于评估项目的主观资源。除此之外，还可以参考一些客观的趋势。

- 社区兴趣

  了解与解决方案相关的用户组和聚会的数量（及用户的参与程度），这可以很好地衡量人们对技术的感兴趣程度。

- 谷歌趋势

  谷歌趋势也可以很好地衡量人们对项目或技术的感兴趣程度。

- GitHub 活动

  GitHub 可以体现人们对项目是否感兴趣以及项目的活动情况。例如，可以了解项目的贡献者数量、提交数量等。一个蓬勃发展的活跃项目会有很频繁的活动。如果某个项目在 GitHub 上看起来很安静，那么就要小心了。

- Jira 问题数量

  除了 GitHub 活动之外，问题跟踪系统（目前用得最多的是 Jira）的活动也为我们提供了有用的见解。例如，查看过去 30 天的问题数量，无论这些问题是处于打开状态还是关闭状态，都可以说明问题。只需要访问开源项目主页，并查看项目过去 30 天的活动。通常，活动越多意味着人们对项目越感兴趣，对项目提供的改善和功能也越多。你还需要了解有多少公司参与了项目，从而真正了解谁在使用这些技术以及提交者的多样性。

- 电子邮件列表和社区论坛

  任何活跃的项目都会有活跃的电子邮件列表或在线论坛。留意这些方面的活动，它们的活跃度是在增长还是在下降？

- 技术大会和用户组

  蓬勃发展的项目都会举行相关的活动，人们在这些活动中讨论项目的情况。这样的活动可能是会议、专门的大会，或者活跃的用户组社区。会议通常由供应商主持和赞助，如果有可能，最好多参加由开发人员和用户发起的聚会，因为供应商总是试图借助大会推广自己的产品。

- 贡献者

  谁在为项目做贡献？理想情况下，贡献者应该是一个多元化的群体，成员来自多家公司。如果只有一家公司在项目开发中占主导地位，那么就要谨慎地评估这个项目。

- 资金链

  最后一个方面是资金链。参加技术大会的人多吗？如果公司已经上市，那么股票的走势如何？市场上是否有竞争对手，它们的发展状况是怎样的？

在缩小选择范围后，需要考虑需求、内部技能水平、风险承受能力以及发展意愿等因素。这些因素是面向团队的——你需要一个能够让员工和公司感到自豪并乐于参与的解决方案。

在全力以赴开始项目之前，先对解决方案进行测试。请记住，不一定要局限于一种选择——可以同时评估多个解决方案，比较它们的初始进度。如果选择的解决方案不适合你的应用场景，或者在开发过程中遇到了挑战，可以改变方向，寻求其他解决方案。

2.5 小结

与其他技术选型过程一样，选择大数据解决方案需要先定义以下关键项。

- 项目的需求是什么？
- 公司已经掌握了哪些技能？
- 公司具备多大的风险承受能力？
- 有怎样的时间表和预算？

无论是评估开源解决方案还是商业解决方案，厘清这些问题都有助于更好地做出决策。此外，在选择商业解决方案时使用的资源（例如分析师、会议和同行）同样适用于开源项目。

不过，在开源世界中，为了能够彻底评估特定的解决方案，了解项目的生命周期是非常重要的。此外，与选择商业解决方案一样，能够在做出决策时批判性地评估供应商的承诺和基准测试也非常重要。

本章主要关注技术选型过程和通用的考虑因素。第 3 章将介绍如何在选定技术后降低风险。

第3章 数据项目的风险管理

人类总是担心这担心那，但通常都是杞人忧天。当特德还是孩子的时候，他浪费了很多时间计划如何避开僵尸的攻击，而这些时间本应该用来睡觉。人们总是担心很多事情，无论是现实的还是臆测的，而这些事情可能永远不会给他们带来任何影响。例如，人们害怕遭到鲨鱼攻击或恐怖袭击，但实际上，死于心脏病或车祸的概率更高。

软件开发也是如此。在开发软件项目时，特别是在项目中使用新技术时，搞清楚应该担心什么和不应该担心什么是非常重要的一步。如果能够走好这一步，风险可能就会变成机会——如果在一开始就了解项目的一切，那么还有什么乐趣?

本章将讨论在完成技术选型并进一步实现项目时如何管理风险。本章的重点是帮助你建立取得成功的方法论和环境，你将学会使用开发原则和策略来管理和降低风险，设定切合实际的期望以及组建成功的团队。

3.1 风险类型

在深入探讨风险管理的细节之前，先来看看需要在项目规划阶段解决的各类风险。

3.1.1 技术风险

任何软件项目都存在风险，而构建大型、复杂的分布式数据解决方案面临的风险会更多，在基于不熟悉的新系统构建项目时尤为如此。风险可能来自架构中使用的各个组件以及组件之间的交互。

对设计系统时使用的技术不熟悉也是风险的潜在原因。所幸的是，我们可以采用一些策略来管理和缓解这些风险。

3.1.2 团队风险

团队风险是指与实现数据解决方案的内部团队和外部团队相关的风险。这种风险可能来自内部团队的知识和实力水平不足、对外部团队的依赖，甚至可能来自喜欢制造混乱的团队成员。

3.1.3 需求风险

需求风险通常来自定义不明确的需求或问题，或者因为团队之前没有处理过某些类型的需求（特别是在采用新技术时）。例如，延迟或吞吐量需求超出了团队的处理能力。需求范围蔓延也是需求风险的常见来源。

下一节将详细介绍这些风险类型，并讨论如何管理它们。

3.2 风险管理

本节将从一种方法展开讨论，该方法根据一种高级模型为系统组件分配不同的风险级别——这在各种项目的实践中都很奏效。这些风险级别包含 3.1 节列出的风险类型，并提供一个用于量化和消除这些风险的框架。

3.2.1 对架构中的风险进行分类

这种方法首先将架构分解为多个部分。图 3-1 展示了如何拆分常见的数据存储和处理系统。

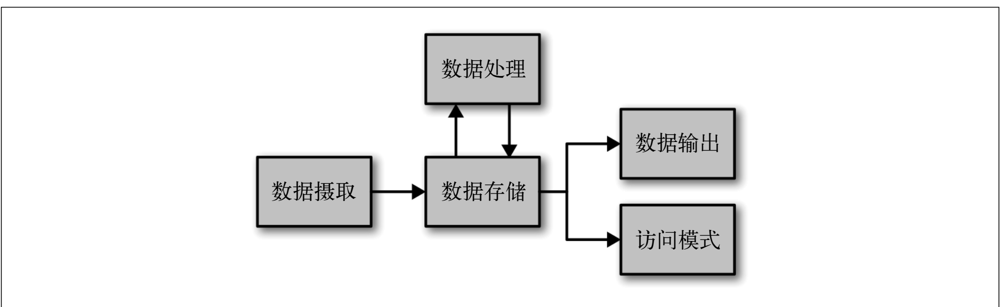


图 3-1：将系统拆分为高级组件

在图 3-1 中，我们将系统拆分为数据摄取、数据存储、数据处理、数据输出和访问模式。可以进一步将这些组件拆分为更多的子组件，如图 3-2 所示。

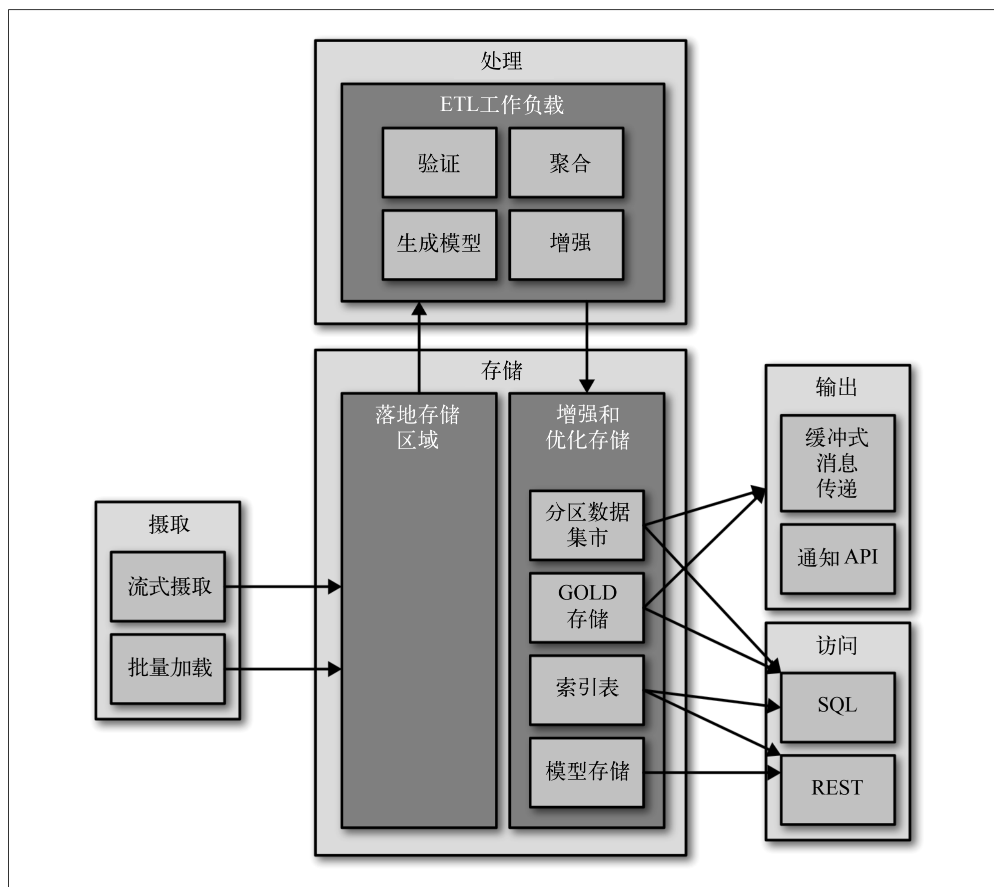


图 3-2：将系统拆分为子组件

到了这个阶段，我们通常不再进一步拆分系统。在达到这个级别之后，要为每个子组件定义接口（3.4 节将讨论接口设计），这样做有助于降低发生故障的子系统对系统其余部分造成影响的风险。现在，只需要将整体看成是一个包含很多独立组件的系统，你可以单独开发这些组件，将风险限制在每个组件的内部。

下一步是确定用来实现各个组件的技术，如图 3-3 所示。然后，指派团队成员开发各个组件，并评估技术和开发团队的风险权重。我们将从技术风险开始，简短地讨论一下这个过程。

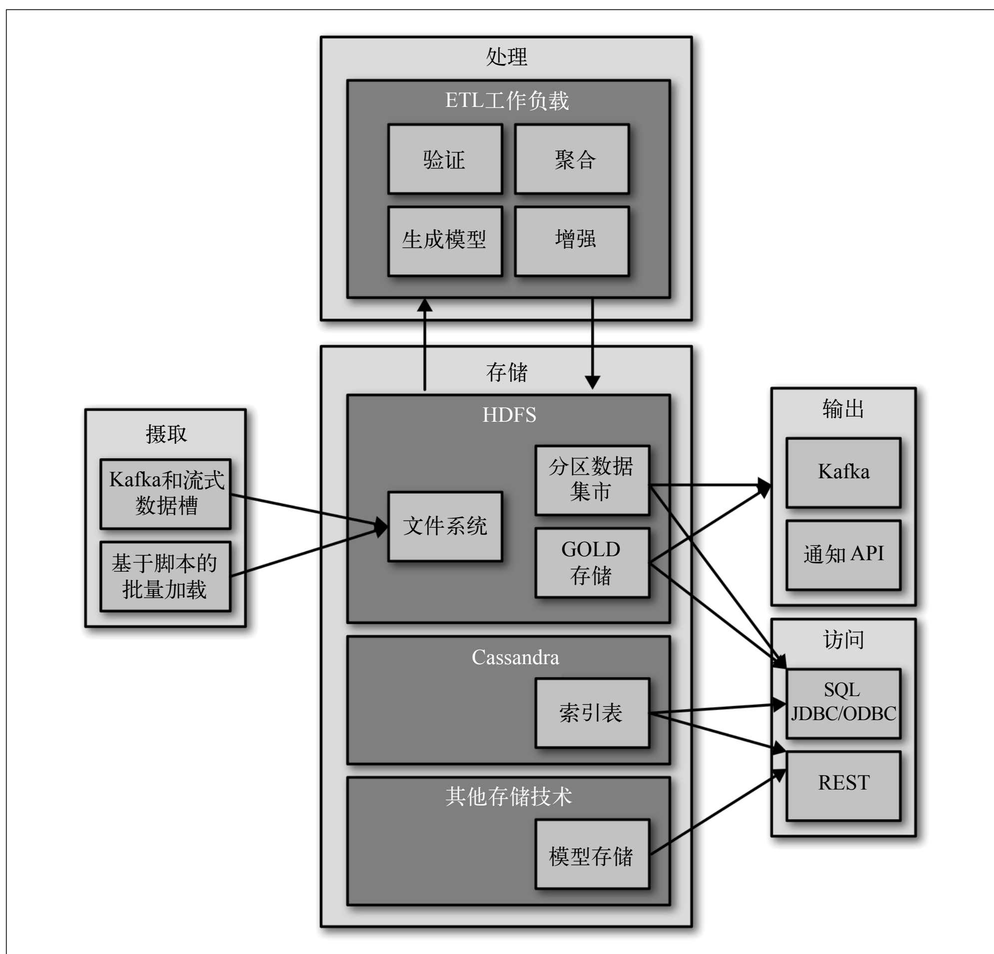


图 3-3：确定子组件所使用的技术

图 3-3 表示已经完成了技术选型，其中使用了以下技术。

- Kafka

  用于缓冲数据流。

- HDFS 和对象存储

  用于数据的初始化加载和长期存储。

- Cassandra

  用于高性能索引存储。

- 其他

  内存数据库和基于搜索的系统（如 Elasticsearch）。

以上技术选型示例只作说明之用，后文将深入探讨选择某种技术的原因。举例来说，第 5 章将详细讨论存储系统，第 8 章将讨论数据处理系统。要点是，在初始设计阶段就选好技术，如果某项技术不合适，就将它替换掉。接下来看一个例子。

假设选择了一个 NoSQL（例如 Cassandra 或 HBase）作为时序数据库的基本存储。刚开始一切都很顺利，但在项目进入后续阶段时，就会碰到如下的问题。

- Cassandra 无法在任意给定的节点上存储足够多的数据，那么存储成本就会成为一个问题。
- 处理数据（例如压缩数据）的 HBase 或 Cassandra 后台进程开始影响性能。
- 在某个度量指标达到一定的数量时，聚合查询的成本会变得非常高。
- 性能方面没有问题，但供应商的定价会成为一个问题。

因此，初始技术选型不奏效可能有多种原因。所幸的是，如果遵循第 4 章提供的有关在架构中使用接口的建议，就可以在不对现有设计造成重大影响的情况下做出变更。

3.2.2　技术风险

给团队或技术设定风险权重并不是一门科学，通常可以根据直觉来判断。如果某项技术之前在生产环境中使用过，并且团队拥有丰富的经验，那么可以给这项技术设定较低的风险等级。相反，如果对某项技术几乎没有使用经验，那么应该设定较高的风险等级。

此外，了解一项技术通常会涉及多个层面的知识。以 SQL 为例，团队可能拥有多年的 SQL 经验，但对于在较新的“大数据”查询引擎（如 Hive、Spark SQL、Cassandra CQL 或 Impala）上使用 SQL 没有什么经验。并非所有的查询引擎都具有相同的行为，而且不同的引擎可能支持或不支持特定的 SQL 功能。了解这些功能和限制是用好这些系统的关键，有助于进一步学习除查询语言之外的知识。当我们说自己了解某项技术时，应该已经彻头彻尾地了解它。

有时候，最好的工具就是已经知道的那个

请记住，一个问题几乎总是可以用多个工具来解决。正如第 1 章所讨论的那样，你通常可能已经在使用这些工具。如果有多个组织（包括你的组织在内）已经通过某个工具获得了成功，那么就可以安全地将它用在架构中。

关键在于团队对工具和项目需求的了解程度。这有助于了解该工具是否适合项目。锤子一般用来将钉子钉入墙壁，虽然也可以用它来开罐头，但那绝对不是它所擅长的，用它来擦窗户也不合适。

简而言之，了解工具及其用途是成功使用工具的关键。

3.2.3　团队的优势

团队的风险等级取决于个人对团队成员能力的了解程度，以及团队过往完成任务的情况。

团队成员的经验越少，团队的风险权重就越高。

当然，无论经验水平如何，每个团队都会有不同类型的人才，他们掌握着不同类型的技能。以下描述了一些典型的开发团队类型，一个全面的团队应该至少具有其中某些角色。

- 清道夫

  清道夫对细节一丝不苟，确保项目具有完整的测试覆盖率，让代码完全处于版本控制之中，等等。

- 原型设计者

  原型设计者喜欢研究新软件。他们的职责是测试各种方法，触及风险区域，并及时对设计方案做出修改。

- 主力

  主力扮演着关键角色，因为他们将完成大部分工作。

- 变通者

  变通者渴望学习和成长，适应能力很强。他们可以帮助其他成员与项目一起成长。

- 谈判者

  管理项目以及与外部团队沟通也是重要的任务。并非所有开发人员都擅长这种沟通风格，所以需要既懂技术又知道如何跨团队工作的人。

此外，某些人格类型可能会给团队带来危险，并对项目产生直接的影响。

- “牛仔”程序员

  任何做过技术工作的人都熟悉这类开发人员——他们更愿意以自己的方式独自工作。他们通常是高效且有才华的程序员，但不擅长合作。他们能够比普通程序员完成更多的工作，但通常不会遵循编码规范或进行良好的文档化。有些时候，这些“牛仔”是一种资产，但他们无法与团队进行良好的合作并遵循标准和流程，这可能会变成团队的负担。

- 有害的个性

  每隔一段时间，团队中就会出现一些成员，他们会在无意之中破坏团队的凝聚力。这可能由多个因素引起——比如这些人好争论，认为自己的解决方案是最好的。矛盾的是，这些人通常也是才华横溢且富有成效的人，但他们对团队士气和生产力带来的影响通常会超过他们的才华给团队带来的好处。所以，要避免这类人出现在团队中——不那么有才华的人或许会更好，只要他们能够很好地融入团队。如果无法避免，那么就要学会如何管理他们。

此外，最好可以拥有足够多的团队成员，这样每个组件就可以至少由两个人负责。让团队成员交叉负责其他组件，不仅会带来更好的组件覆盖率，通常还可以获得更好的结果。

3.2.4 外部团队风险

除了与项目团队相关的风险之外，在与一个或多个外部团队沟通时也存在风险。如果这些外部团队开发的组件或系统对项目团队至关重要，或者反过来，项目团队的工作对外部团队至关重要，那么这种风险就尤为明显。如果你的成功直接与另一个团队挂钩，那么就要保持警觉和尊重。明确责任和需求有助于实现这一目标。此外，在设计需要与外部系统发生交互的软件时，要清晰地定义接口并将它们文档化。



本章及第 4 章都会详细讨论接口。

3.2.5 需求风险

在讨论过团队风险和技术风险之后，接下来讨论需求风险。这种风险具有不同的表现形式。其中一种是定义不明的需求，这意味着后续需要进行调整，而这些调整会引入新的风险。另一种是团队之前未曾遇到过的需求，例如服务等级协定（service level agreement，SLA）和延迟指标。

明确地定义功能性需求有助于降低这些风险。如果将功能性需求视为非技术性的合约，就可以有效地让项目保持在正轨上。本章稍后将讨论其他可用于降低需求风险的方法。

将需求分解为更易于管理的工作块是降低需求风险的另一种方法。举个例子，有个客户向本书的一位作者展示了一份上百页的系统组件需求文档。这份文档来自一家咨询公司，用来帮助客户实现解决方案。事实证明，咨询公司的做法失败了。这份需求文档采用的方法有点好高骛远，导致客户不堪重负，不知道该从哪里着手。尽管文档记录了他们的需求，却仍然无助于确定切实可行的开发时间表以及需求和任务的优先级。最后，这份文档被弃用了。

在弃用这份需求文档之后，他们开始定义高级需求。这些需求的内容可以被塞进单页文档中，并附带一幅解决方案架构图。图 3-3 给出了这个架构图的示例。其中定义了一个非常重要的组件，如果实现了这个组件，就可以提升对解决方案的信心。在这个有限的范围内，可以针对所选组件进行概念证明（proof of concept，PoC）。根据从 PoC 中获得的见解对高级需求进行调整，然后识别出另一个组件，以便进行下一个 PoC。这个过程不断重复，大概一个月就可以粗略完成整个系统的 PoC。

其结果就是识别出了风险，开发团队对需求有了更深层次的理解，客户对设计更有信心，并且根据从 PoC 中获得的见解估计出实际的系统部署时间表。

3.2.6 融会贯通

为了理解这一切是如何与创建的系统模型相匹配的，先来回顾一下之前讨论过的风险类型。

- ❑ 技术风险

  团队有哪些使用软件开发应用程序的经验？

- ❑ 团队风险

  团队的经验、团队里有哪些类型的成员、对其他团队的依赖，等等。

- ❑ 需求风险

  需求有多大风险？

1. 分配风险权重

在分析了不同的风险类型后，可以根据分析结果为创建的系统模型分配不同的风险权重。如果将所有风险权重考虑在内，应该会得到如图 3-4 所示的结果。请注意，这张图在印刷后是灰度图[1]，最好在图中使用不同的颜色来突出显示不同的风险级别。

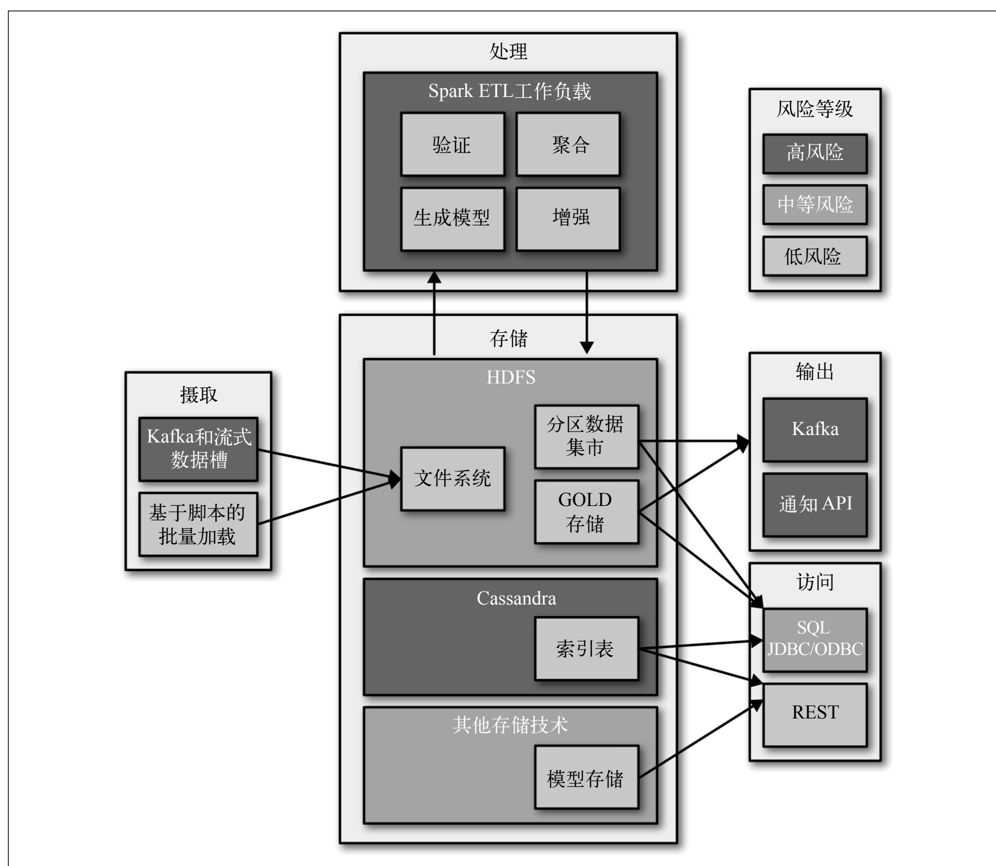


图 3-4：分配风险权重

注 1：本书彩图可到 http://ituring.cn/book/2641 查看。——编者注

深色表示风险级别较高，浅色表示风险级别较低。

还应该详细解释为什么某些组件被分配了较高的风险级别。列出每一个组件，然后附上组件和相关需求存在的风险。如果每个风险都带有一个简短的标题，就可以很方便地在讨论中引用它们。以下是一些示例，列表中的每一项都标有风险类型。

- Cassandra
  (1) **技术经验**（技术风险）：公司内部的 Cassandra 经验很有限。
  (2) **数据模型**（需求风险）：需要针对用例验证数据模型的正确性。
  (3) **正常运行时间**（技术风险）：根据 SLA 需求，存储在 Cassandra 中的数据必须为用户提供 99.99%的正常运行可用性，这超出了 SLA 的正常处理能力。

- 通过 Spark 批量获取数据
  失败场景（需求风险）：由于存储层有问题而无法提取数据，不知道该如何继续。

- Kafka Streaming
  (1) **技术经验**（技术风险）：团队第一次构建流处理应用程序。
  (2) **零数据丢失**（技术风险）：要求零数据丢失，因此引入了非常复杂的技术解决方案。

- Spark ETL
  (1) **资源可用性**（团队风险）：具有 Spark 经验的团队成员都已经满负荷了，这意味着需要将工作分配给经验不足的团队成员。
  (2) **数据模型**（需求风险）：数据模型定义的不确定性意味着可能需要重新实现代码。

2. 最大限度地降低风险

在评估完风险级别之后，接下来采取措施将风险降至最低。我们之前已经讨论了一些步骤，接下来通过一个列表进行更详细的介绍。

- 更好地锁定需求，定义详细而精确的功能性需求。
- 传达需求并获得所有相关人员的认可。
- 明确定义项目的范围并得到相关人员的同意。
- 为外部提供接口需求和协议（稍后会更详细地讨论这方面的内容）。抽象有助于降低架构风险，因为它让组件的重新实现或者在必要的情况下替换现有技术变得更加容易。
- 优先考虑风险较高的事项，为解决问题提供额外的时间。
- 除了上述的建议外，可以将风险较高的技术问题分配给更有能力、更有经验的团队成员。
- 借助外部资源来弥合知识方面的差距（本章稍后将介绍更多这方面的内容）。
- 使用原型和 PoC 来降低架构风险。
- 使用风险较低的组件替换风险较高的组件。例如，在假设的风险细分场景中，我们可能会用 MapReduce 或 Hive 来代替 Spark ETL，因为团队成员对前者更熟悉。

3.3 使用原型和PoC

架构中的某个组件可能无法达到预期，所以有必要制定备用计划。但组件故障会影响对项目或计划的看法。如果在故障对项目产生进一步影响之前就让它失效会怎样？下面介绍一些方法。

3.3.1 找到两三种方法

诀窍很简单：从影响范围较广的需求开始，通过头脑风暴讨论出两三种方法来满足这些需求。实现每一个解决方案，然后进行基准测试比较。这样做有助于选出最佳的解决方案，或者进一步尝试其他解决方案。关键是速度要快——可以将这些解决方案视为原型。这个过程包括以下要素：

- 系统的文档质量；
- 解决方案的性能，如吞吐量、延迟等；
- 不同方法的复杂性；
- 团队成员掌握新技术的能力以及他们对技术的看法。

3.3.2 进行PoC，然后丢弃

PoC 是验证技术或方法的一种有效方式。PoC 应该被视为一次性工作，并尽快地完成。PoC 的目标是在有限的时间内尽可能地推动需求和技术向前发展。在 PoC 成功之后，需要重写代码，甚至需要让不同的开发人员重新实现大部分内容。这将带来以下好处。

- ❑ *更多的视角*

  不同的视角可以更好地验证 PoC。

- ❑ *更好的系统*

  从 PoC 中获得的经验和知识通常有助于构建出更好的系统。

在进行 PoC 时通常需要避免一个问题，即管理层在看到 PoC 结果时会说："它奏效了！赶紧发布吧。"话虽如此，但 PoC 应该被视为生产版本的最小可行产品，从而更好地评估解决方案的适用性。

3.3.3 部署的注意事项

在以前，代码通常被部署到手动配置的服务器上，需要根据很长的检查清单来升级或搭建一个新系统。在部署新代码时，缺乏可靠且可重复的变更管理流程将带来相当大的风险。成熟的企业通常会使用自动化软件来管理和执行系统变更，这些自动化软件包括构建系统（如 Jenkins）、配置管理系统（如 Puppet 和 Chef），以及容器（如 Docker）。通过使用这些系统可以极大降低软件部署的风险。

深入探讨这些系统超出了本书的范围，不过，这里要说的是，可靠且可复用的构建系统和部署系统是快速推进项目并降低风险的关键。生产环境的很多问题是由升级、配置变更、库更新等操作引起的。构建和部署的自动化程度越高，在生产环境部署过程中发生问题的风险就越低。

3.4 使用接口

接口是软件开发中的一个常见且重要的概念。它可以减少组件之间的耦合和依赖，并且降低软件架构风险。项目级别或架构级别的接口设计是指：让应用程序的不同部分与架构之外的区域之间的通信模式达成一致。这个概念的常见实现是将服务层作为前端 Web 应用程序和后端数据存储之间的接口。可以通过多种方式来实现接口：描述性状态迁移（representational state transfer，REST）接口、Java 接口、Scala trait，等等。



与接口相关的主题非常重要，所以第 4 章将单独讨论如何设计和实现灵活且可维护的接口。不过，现在先简要介绍一下接口如何帮助我们降低项目风险。

接口是如何降低风险的呢？一种方式是让团队单独开发系统的不同部分。例如，前端团队构建接口的虚拟实现，从而继续开发、测试和部署他们的 Web 应用程序，而无须关心后端团队的进度。图 3-5 展示了这个过程。

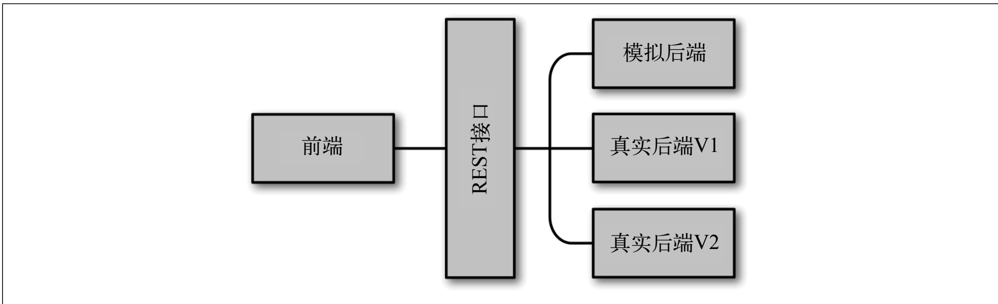


图 3-5：接口的模拟实现

这也意味着后端团队可以自由地在实现中改变存储或执行引擎的策略，而不会影响前端团队的工作。

使用精心设计的接口可以让前端团队和后端团队自由地修改各自的设计和实现方式，而无须咨询对方，只要遵循定义好的接口即可。各个团队可以独立地进行编码、负载测试、单元测试等，不需要相互依赖。这种方法的另一个好处是可以显著地缩短开发时间。

良好的接口设计还有助于降低沟通风险，因为它将系统分成了多个部分：

- 前端；
- 数据摄取；
- 流处理；
- 存储；
- 数据库；
- 批处理；
- 实时处理；
- 监控；
- 审计；
- 灾难恢复。

一个组件自身的风险几乎不会对其他组件产生影响，这有助于管理风险。风险被隔离到单独的组件中，而不是在整个项目上。当某个组件发生故障时，就可以体现出这种设计方式的优势。首先，发生故障的组件不会影响其他组件的进程。其次，当故障组件被隔离时，只要接口设计保持不变，就可以自由地修改这个组件，甚至完全重写它。

3.5 尽早开始构建

公司容易掉入一个陷阱——在所有需求都明确之后才开始开发软件，这通常是因为担心过早开发的代码会被丢弃，金钱和时间会受到损失。事实上，尽早开始开发反而会带来一些好处。

- 提高开发团队的参与度，让他们更好地了解需求，并在收集需求的过程中发挥更积极的作用。
- 尽早清除需求中的问题。
- 可以提供演示，这非常重要，原因有二：一是通过向外界展示项目进展来降低项目风险；二是从用户那里收集反馈，用户看到演示后通常会说“它跟我想的不一样”。
- 如果出现问题，可以通过多个方案来解决，并更好地了解事情的来龙去脉。

3.6 频繁测试并保留记录

通常，项目风险可以用数字进行量化，例如响应时间、并发量、吞吐量、SLA、正常运行时间和故障转移时间。要确保记录了这些指标，并进行定期的检查和评审。测试和重新测试是这个过程的重要组成部分。

假设有一个处理过程涉及批量加载操作，需要测试吞吐量和总执行时间。为了尽早完成测试，团队使用数据生成器来创建输入的测试数据。基于生成数据运行的测试最终得到了很好的结果，团队利用这些结果来评估整个项目。但团队并不知道，生成数据的熵非常低（熵可以衡量数据的随机性）。更高的熵意味着更差的压缩比，而更低的熵意味着更好的压

缩比。在本例中，低熵使得执行引擎的压缩编解码器更快地读取数据和写入数据，并且更快地通过网络发送数据。

因此，当团队开始使用真实数据进行测试时，他们惊讶地发现作业的运行速度比初始测试时要慢得多，而且资源占用率也很高。

从这个例子中可以学到以下几点：

- 尽早并频繁地测试；
- 仔细记录测试的内容；
- 记录测试数据与真实数据之间的差距；
- 让测试尽可能反映生产环境的情况；
- 对任何度量指标都持半信半疑的态度；
- 始终假设得到的测试结果好得有点不真实；
- 让不止一组人来评审测试流程、基准测试，等等。

在向项目以外的人传达测试结果时，需要考虑到以上所有事项。你很难向他们解释为什么今天的处理速度会是昨天的十分之一——如果试图在事后解释测试结果之间的差距，人们一般很难认为这是情有可原的事情。

3.7 监控和警报

本章已经花了大量篇幅介绍如何规划和管理架构中的风险，接下来谈谈在部署阶段如何管理风险。需要找到一种方法来指定和跟踪指标，以确保系统在部署后能够按照预期的方式运行。为此，我们为架构中的每个组件都定义了关键绩效指标（key performance indicator，KPI）。每个系统的 KPI 应该会有所不同，但一般来说，至少需要 3 个 KPI。《SRE：Google 运维解密》一书对这 3 个 KPI 给出了定义。

- ❑ 吞吐量

  吞吐量可以告诉我们系统正在做什么以及可以做到什么程度。

- ❑ 延迟

  系统执行给定的操作所需的时间。

- ❑ 错误率

  给定的操作发生了多少错误。

如何定义 KPI 和构建监控系统已经超出了本书的范围，但有很多参考资料（例如前面提到的图书）提供了细节。这里要强调的是，如果没有一种好的方式来监控系统，就像是在蒙着眼睛走路。在项目经过实现阶段并进入生产部署阶段之后，定义这些 KPI 并构建适当的监控系统将在降低项目风险方面起到重要的作用。

3.8 沟通风险

只在口头上讨论风险，这本身就是一种风险。如果过分强调风险的现实性，那么项目之外的人会对项目产生担忧，导致不必要的人员介入。在一家大公司开发软件就像在厨房里做饭，不必要的人员介入就像是让 5 个厨师同时制作一个三明治。此外，如果涉及办公室的勾心斗角，事情会变得更糟糕：机会主义者可能会夺走你的项目；反对者有理由质疑设计中的每一个小问题。这会浪费很多时间，给团队带来负面影响，并损失团队的名誉。

低估项目风险也是一种风险。你可能需要寻求帮助，而且如果因为低估了风险而在寻求帮助之前等待太久，可能会错过最后的期限，导致系统崩溃或数据丢失。结果就是你丢了项目，其他人丢了工作，长期以来在公司的努力化为泡影。

因此，不难看出，如果未能以正确的方式向外传达风险，最好的结果是只对项目造成伤害，而最差的结果就是丢了饭碗。在向外传达风险时，沟通和时机至关重要。本节将讨论一些策略，一方面有助于传达风险，另一方面能够保护项目的进展。

3.8.1 合作并获得信任

要点是通过与组织中的其他人员展开密切合作来降低风险。你可以将设计理念抛给其他人，让他们贡献想法。这可以获得多种好处。

- 拥有第二（或第三、第四）双眼睛

  从外部获得的额外反馈通常可以提供自己想不到的见解。无论经验和知识水平如何，从不同的角度看设计都会带来好处。

- 获得信任

  通过采用他人的设计或想法来获得支持是一种非常好的方法。当别人看到你在设计中采纳了他们的想法时，他们会认为自己为你的成功做出了贡献。

- 公开讨论风险

  如果将风险当作具有挑战性的问题来解决，不仅可以得到其他人的帮助，还可以让高层管理人员解决超出你控制范围的问题。

- 保持可控并展示进展

  每个项目都会存在问题和风险，而谈论风险和传达解决风险的进展是在向其他人展示团队解决问题的能力。这样做可以增强团队应对风险的信心。

3.8.2 公开风险

如前所述，与第三方（如咨询顾问、供应商或来自其他部门的可信赖成员）沟通风险有助于进行风险管理。假设第三方在项目要解决的问题和架构方面拥有丰富的经验，他们可以

通过多种方式提供帮助。

- 如果他们之前构建过类似的系统，就可以帮你避开陷阱，节省时间和金钱。
- 帮你做出不受办公室不良风气影响的决策。
- 如果他们是对的，并使项目取得了成功，你应该会很开心。

第一点可能是最重要的。现实情况是，大多数项目需要借助其他公司构建的解决方案。除非身在谷歌、亚马逊或 Facebook，否则项目可能只是对第三方资源千篇一律的实现。如果真的想降低风险，那么获得专家的帮助通常是一种可靠的方法。

3.9　将风险作为谈判工具

最后，将项目风险作为谈判工具，这一点也至关重要。这个概念似乎有点违反直觉——为什么要使用风险以及如何使用风险作为谈判的手段？事实上，有时候利用风险可以促进项目管理，例如争取更多的资源、更改项目范围或时间表。不过，在选择将哪些风险用于谈判以及如何在谈判中利用这些风险时，务必要谨慎。

适合用于谈判的是那些与业务需求有密切联系的风险，它们可以为谈判提供坚实的基础。

例如，业务方要求存储 TB 级的数据，并且能够在几毫秒内访问数据；但公司只有 MySQL，显然并不适合这种用例。假设公司为了解决这个问题决定使用 NoSQL，但开发团队在这方面没有太多经验。缺乏经验是一种风险，但它是由业务需求直接带来的。

因此，如果存在与业务需求相关的风险，可以在规划项目时要求如下事项：

- 额外的项目时间；
- 额外的资金，用于咨询第三方专家；
- 更多的预算，用于招聘、培训等方面。

需要注意：如果管理层提供了所需的资源，他们认为风险就会随之消失，这会导致他们的期望值变高。此外，如果不断提出请求，业务方可能会怀疑你的能力或团队交付解决方案的能力。因此，要明智地提出请求，并了解清楚完成项目所需的资源。在最糟糕的情况下，业务方可能会因为感觉到失败的风险而取消项目。

因此，在进行谈判时，请搞清楚可以请求什么以及需要什么。尽可能保持诚实，因为无论怎么撒谎，最终都会让你一败涂地。

3.10　小结

本章介绍了数据项目的风险管理，以及如何消除和最小化风险，几种风险类型如下。

- ❑ 技术风险

  与技术选型、如何使用这些技术来创建解决方案等因素有关。

- ❑ 团队风险

  来自项目团队和外部团队的风险。

- ❑ 需求风险

  来自定义不明确的需求或者团队之前没有遇到过的需求，等等。

然后，本章介绍了消除或降低这些风险的方法。

- 为系统建立高级模型，根据这些风险类型的组合来分配风险级别。
- 消除技术和架构风险，包括使用抽象和 PoC 等。
- 组建强大的团队，并与外部团队合作，一起应对挑战。
- 定义可靠且易于管理的需求。
- 利用外部资源帮助项目取得成功。

本章还讨论了沟通风险以及将风险作为谈判工具。确保已经将风险记录下来，并为管理和降低风险制订计划，这些对数据项目的成功至关重要。

第 4 章

接口设计

使用海量多结构数据构建数据项目是一项非常复杂的任务，在很多方面存在潜在的复杂性，例如将各种组件和技术集成到解决方案中，支持大量的并发用户和不同的处理模式，以及适应不断变化的需求和技术。

在设计系统时，应该如何应对这些挑战？经过良好设计的接口能够帮助我们构建出可维护、可扩展且可适应变化的解决方案。本章将重点介绍什么是好的接口设计，还会介绍一些非功能性考虑因素以及一些常见的接口示例。

4.1 人体

在讨论技术之前，先来看一个比我们能构建出的系统更复杂的系统。这个系统同样使用接口和目的性非常强的子系统来减少依赖，并随着时间的推移适应不断变化的状况（不过，它需要处理的时间跨度比我们构建的系统所要处理的更大)。这个系统就是人体。如果把人体看成由接口连接起来的高级部件的集合，那么就可以将它与我们要构建的系统进行类比。

4.1.1 人体与数据架构

如果深入探究人体的主要系统，就会发现它与现代数据架构有一些有趣的相似之处。让我们从人体的周围神经系统和中枢神经系统谈起。

1. 周围神经系统

周围神经系统（peripheral nervous system，PNS）是连接人体各部位的信息管道网络，用来在人体部位之间发送和接收信息。它横跨我们的身体，接收来自感觉器官的信息。PNS 将信息传递给大脑，并将命令发送到其他系统（如肌肉），指示它们做出动作。可以把它想象成身体的信息高速公路。

分布式数据系统与 PNS 有很多相似之处，最明显的例子就是像 Kafka 这样的**发布－订阅**模型。本章后面将详细讨论发布－订阅模型。现在，可以先将该模型简单地定义为一个系统，它具有向中央代理发布数据的发布者和处理这些数据的订阅者。Kafka 在数据架构中的作用如图 4-1 所示。在这个示例中，Kafka 接收来自外部流或传感器的信息，并将这些信息发送到存储系统或准实时（near real time，NRT）的处理系统。在该系统中，我们可以基于数据做出决策。然后，这些决策会通过 Kafka 返回，并发送给可以根据决策命令采取相应行动的服务。

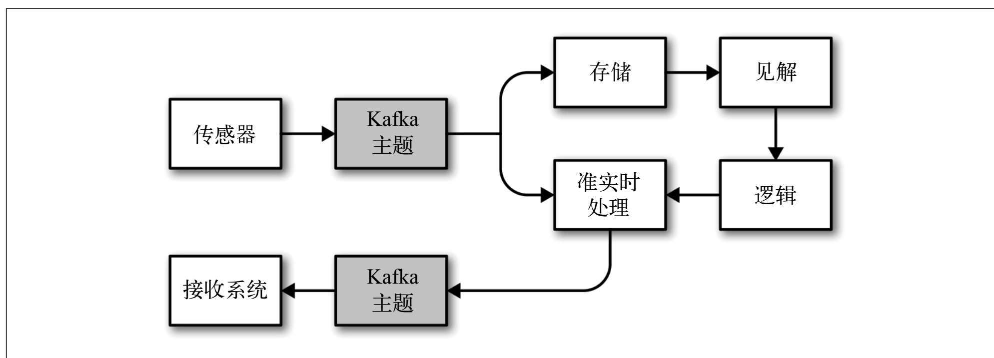


图 4-1：数据架构类比周围神经系统

2. 中枢神经系统

如图 4-2 所示，中枢神经系统（central nervous system，CNS）由神经系统中更有目的性的部分组成，包括脑和脊髓。这些系统所做的事情不只是传输信息，可以进一步将其细分为具有特定功能的子系统。脑由不同的脑叶组成，例如负责视觉处理功能的枕叶、控制记忆和语言等功能的额叶等。这些系统还控制着人体的其他系统，例如肌肉和心脏。

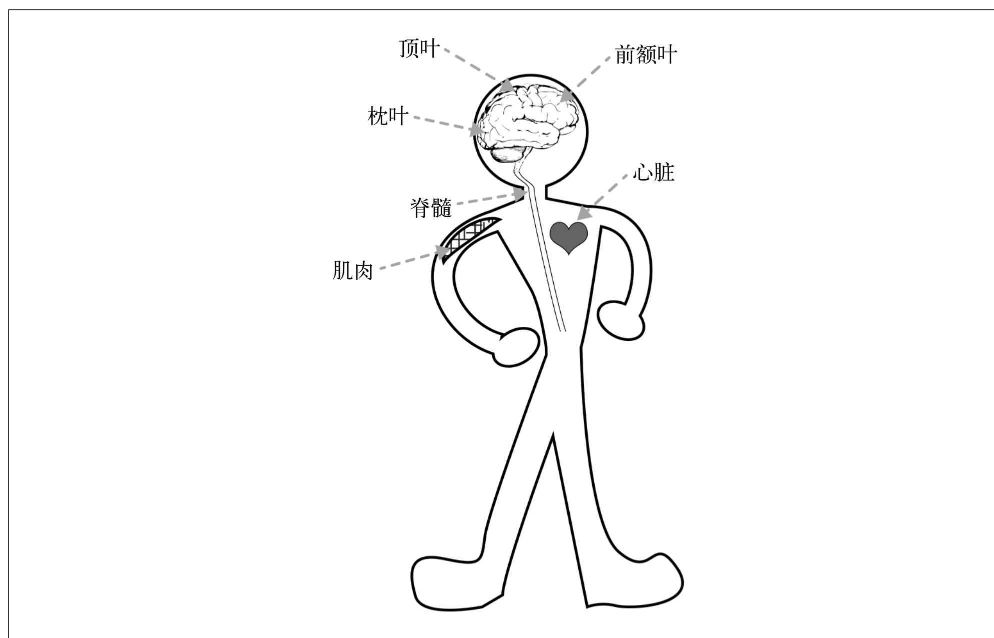


图 4-2：中枢神经系统

如果将其与大数据架构联系起来，那么脑就是图 4-3 中的深色部分。具体地说，脑代表了存储、见解、逻辑和准实时处理组件。

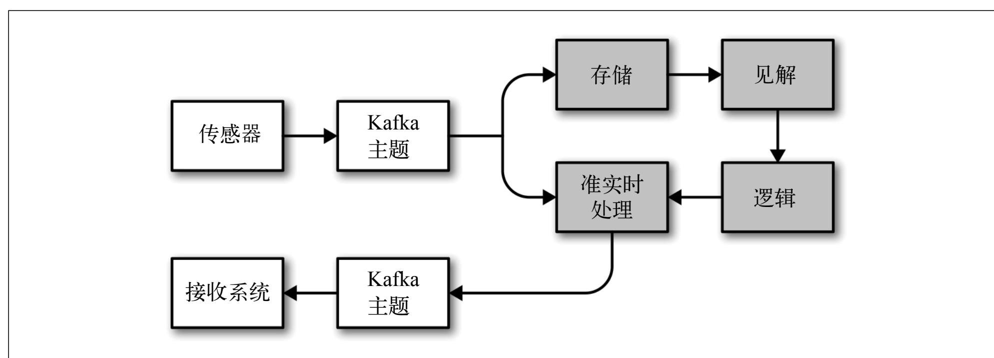


图 4-3：数据架构类比中枢神经系统

接下来花一点时间深入了解这些系统以及它们与脑之间的关系。

- *存储*

  正如脑有长期记忆和短期记忆一样，数据架构也有很多具有不同存活时间的数据存储系统。另外，这些存储系统可以通过多种方式进行索引，就像脑使用不同的模式来存储和检索信息一样。

- 见解

  可以将见解比作脑中的反思性思维，它负责处理传入信息并应用分析来推动决策过程。在数据系统中，这与通过 SQL 和机器学习等工具执行的分析活动有关。

- 逻辑

  正如脑可以通过分析传入信息和应用规则来做出决策一样，数据系统通常也包含了具备类似功能的子系统。

- 准实时处理

  这可能与脑对外界刺激做出快速反应和驱动响应的部分有关。甚至有一些动作是在不经意间发生的，我们称之为“肌肉记忆”动作。CNS 中的逻辑越多，系统处理输入的速度就越快。在数据系统中，可以将其与流处理引擎等系统进行比较。在这些系统中，可以执行复杂的机器学习逻辑，并以毫秒级的响应时间做出决策。

3. 感官

眼睛、皮肤和耳朵等感官可以帮助我们收集来自外部世界的信息。对世界的实际感知是在 CNS 中完成的，这使得感官系统变成了一个只关注信息收集的解耦系统。此外，这些感官使用 PNS 来分发信息。

在数据架构中，任何可以产生数据的部分都相当于感官。它们可以是部署在节点上的代理、输入应用程序日志的系统或设备上的传感器。在图 4-4 中，感官就是传感器（深色方框），它将信息发送给 Kafka 管道。

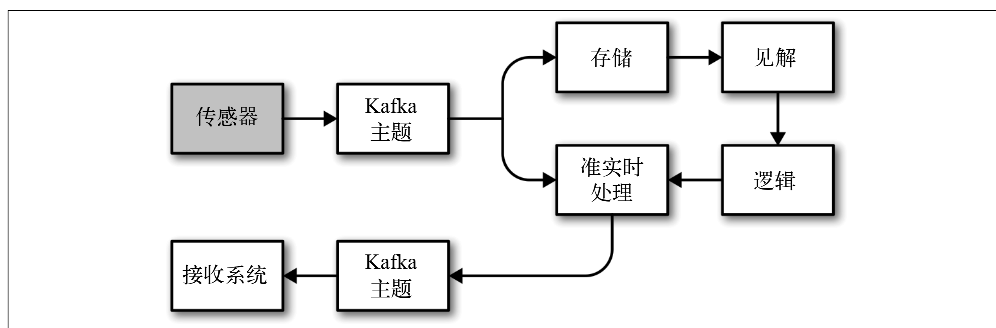


图 4-4：数据架构类比感官

4. 可控系统

某些系统可以通过流经 PNS 的信息来控制。其中，有些控制是有意识的，有些则没有意识。这些系统包括肌肉、心跳和消化系统。

在数据架构中，它们是处理命令的应用程序（这些命令是处理数据得出的结果）。例如，一个面向用户的应用程序，其中一个命令可能是为防范欺诈风险而锁定某人账户的指令。在图 4-5 中，可控系统就是从 Kafka 管道接收输入的系统（深色方框）。

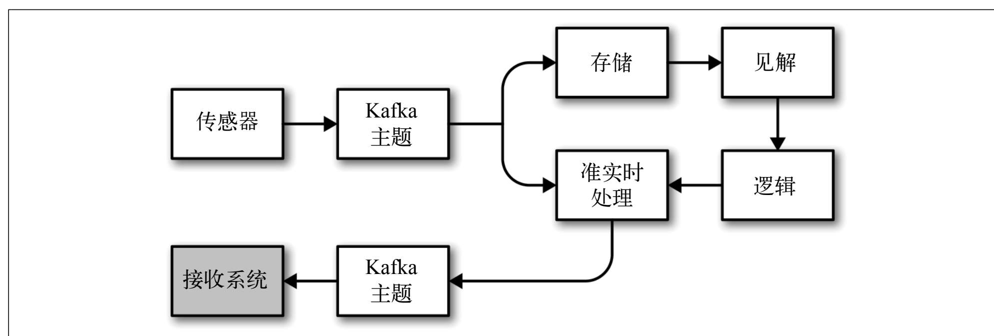


图 4-5：数据架构类比可控系统

5. 人体部位小结

在进一步讨论接口设计之前，先回顾一下人体的例子。需要理解的要点是，不同的系统构成了一个更大的系统。这些系统收集数据、存储数据、处理数据或对数据做出响应。因为存在这些通信，所以需要明确定义接口，并达成一致。这些系统非常复杂，并且独立运行，我们需要以正确的方式对它们进行解耦。对于复杂的数据系统来说同样如此，接下来将更详细地讨论解耦。

4.1.2　解耦

在设计复杂的系统时，**解耦**是一种常见的架构模式——借助组件之间的接口，系统组件可以独立运作。为了与人体进行比较，以心脏和脑为例，如图 4-6 所示。要正常工作，脑依赖于心脏供血，而不是心脏本身。心脏可以被不同的“组件”取代——例如人造心脏——而不会影响血液对脑的必要性。因此，脑与血液（而不是输送血液的系统）之间存在耦合关系。

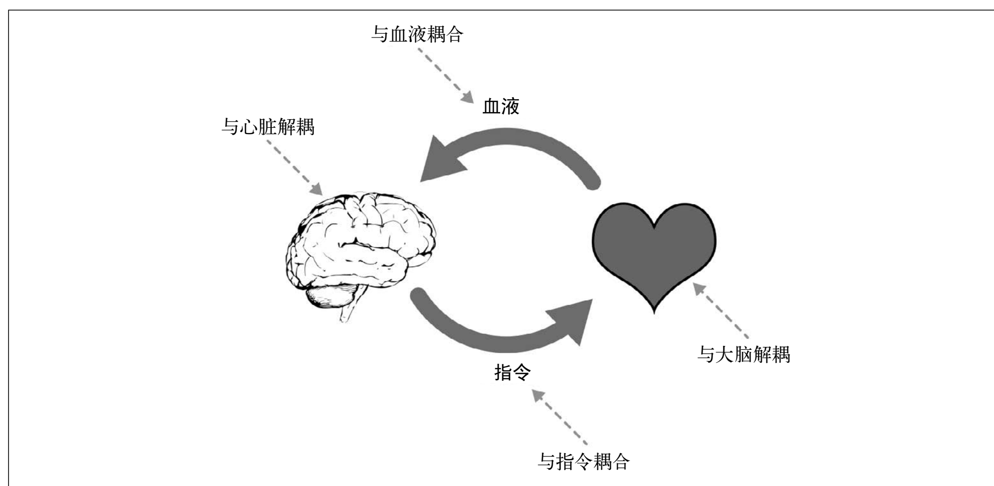


图 4-6：解耦人体系统

解耦可以应用于整个人体。人体由数万亿个细胞和很多子系统组成。子系统通常是独立运作的，它们对人体中其他子系统的运作原理和内容知之甚少。心脏对膀胱一无所知，膀胱也不关心耳朵里负责平衡的器官。脑有一个可以了解其他系统的窗口，但只能通过 PNS 支持的协议来传递消息。脑也是由彼此分离的部分组成，每个部分都有自己的区域和职责。

现代医学开始利用生物学完成一些不可思议的事情，例如绕过损伤的脊柱或直接通过植入大脑的芯片给假肢发送信息，甚至通过一定的电刺激让再生的真肢做出反应。一些实验已经通过嵌入式芯片将摄像头信号发送给大脑，为视障人士带来了有限的视力。

在上面的例子中，视障人士通过摄像头获得视力，大脑并不知道眼睛已被替换为视觉输入系统。大脑中负责解释视觉信息的部分会处理来自摄像头的信息，就好像这些信息来自眼睛一样。

随着系统的增长，软件架构变得越来越复杂，通过接口设计来解耦系统成为了一种非常重要的手段。良好的接口设计可以在不影响系统完整性的情况下添加、删除和开发系统的各个部分。当子系统出现故障时，可以进行修复，并启动新系统来替换。就像生物体一样，架构解耦让我们能够开发子系统而不破坏整个系统的功能。

可以通过各种方式来实现接口设计：分布式消息系统（如 Kafka）、REST、公共 API 以及其他类型的消息（如 JSON、Avro 和 Protobuf）。

为了进一步类比，可以把 Kafka 看成类似于 PNS 和 CNS 的传输媒介，而 JSON、Avro 和 Protobuf 是通过神经发送的消息，如图 4-7 所示。

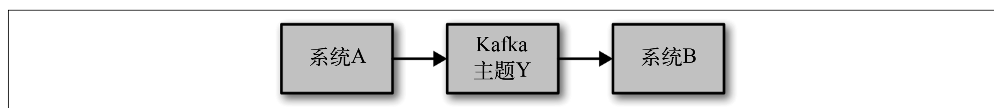


图 4-7：使用 Kafka 解耦

让我们来看看这个 Kafka 架构的解耦选项。

- **隔离**

  如果系统 A 停止工作，它不会影响系统 B，系统 B 会一直等待系统 A 重新上线。反之，系统 B 的故障也不会影响系统 A，因此可以独立测试系统 A 和系统 B。

- **重放**

  如果系统 B 出现故障，可以通过接口和 Kafka 重放数据的功能来重现问题。不仅能测试故障场景，还能验证新版系统 B 能否像旧版那样对数据做出反应。

- **可扩展**

  在 Kafka 架构中，可以在不影响系统 A 的情况下添加更多消费主题 Y 数据的系统，如图 4-8 所示。

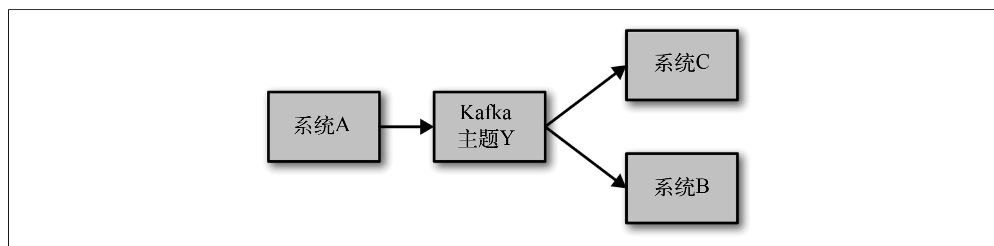


图 4-8：解耦系统的可扩展性

4.1.3 解耦的注意事项

在创造假肢的过程中，大量研究工作涉及大脑向身体的其他系统发送命令的原理。例如，指示手掌张开或合上，如图 4-9 所示。在这项研究完成之前，有科学家认为大脑会向每一块肌肉发送详细的命令，但随着了解的深入，科学家发现大脑发出的命令会在负责处理和执行动作的系统附近被解码。这个过程可以让大脑专注于更重要的事情，而独立系统各自执行命令的细节。随着时间的推移，这些神经通路通过反复暴露于特定的刺激而得到增强。

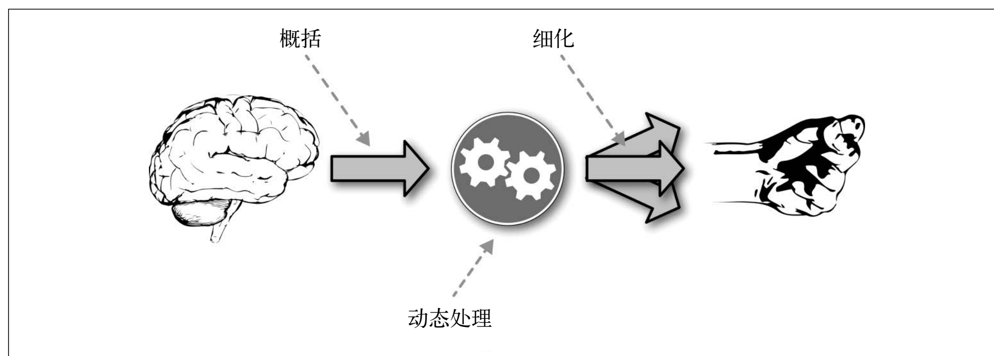


图 4-9：解耦人体系统

可以把这个概念想象成**肌肉记忆**。当做出重复性的动作时，如运动或演奏乐器，练习会带来更好的表现。实际上，你正在优化某些指令的操作和通信，并构建出执行这些操作的子例程或模型。

在分布式数据应用程序的世界里，可以将优化和整合看作从临时处理和批处理到实时处理的转变。临时处理和批处理将始终存在，但它们应该能够让你更接近优化处理。就像形成肌肉记忆可以更好地完成任务一样，在软件架构领域，优化可以降低 SLA、实现更好的资源分配，并促进更高效的处理。所有这些都可以带来更好的终端用户体验。

4.1.4 专门化

最后需要考虑的方面是专门化。以 CNS 和脑为例，脑由很多子系统组成，每个子系统都有自己的区域和职责。有些子系统用于不同类型的存储，有些则用于特定的处理和访问模式。

想想大脑的存储系统。我们有短期记忆、感觉记忆、长期记忆、内隐记忆和外显记忆。为什么会有这么多记忆类型？因为相比只有一个通用系统，拥有多个子系统具有进化方面的优势。这些记忆类型具有不同的索引策略、刷新机制和老化或归档过程。软件架构也一样——不同类型的系统，如关系数据库、Lucene 搜索引擎、NoSQL 存储、文件系统、块存储、分布式日志等，支持不同的存储模型和针对特定应用程序进行优化的访问模式。

大脑的处理系统也一样。视觉解释与复杂的决策是截然不同的。就像大脑一样，软件架构中也有不同的执行模式和优化措施，它们适用于不同的用例，如 SQL、Spark、Spark SQL、NoSQL API、搜索查询等工具。

4.2 什么造就了好的接口设计

在将人体的某些子系统与软件架构中的组件相对应之后，让我们来看看是什么造就了好的软件接口设计。可以将接口视为大型系统的脚手架——如果设计得当，系统可以使用很多年，并且随着时间的推移做出变更，使用不同的技术来实现不同的用例。反之，它就是一个糟糕的设计解决方案。掌握接口设计方法是设计出可靠解决方案的关键。

4.2.1 合约

接口实现者和接口用户之间的合约是良好接口设计的核心。合约必须明确定义接口所定义的每个功能的输入、输出、预期用途和行为。

功能规范是接口的核心，接口的非功能性保证则是次要考虑因素。在某些情况下，需要定义次要考虑因素，比如预期可用性、响应时间、负载容量等——换句话说就是接口所提供的 SLA。本章后面部分将讨论这些考虑因素。

4.2.2 抽象

在定义接口时，实际上是在建立对系统的抽象——系统的实现不需要对接口用户可见。因此，可以将技术选型和具体实现与系统用户解耦。此外，还有多种选项可用于定义接口，接下来看看其中的两个选项。

1. 非编程语言接口

REST 是高级非编程语言接口最常见的示例。一般来说，REST 是一种无状态的客户机 – 服务器端接口，以 HTTP 和 JSON 等通用标准作为基础。

REST 以简单的方式为后端服务提供接口，并可以在不编写代码的情况下调用服务——也就是说，通过 Web 浏览器或命令行调用 REST API。这样就可以非常容易地测试服务调用，为服务创建简单的客户端。使用 Java、Python 和 C++ 等通用编程语言创建客户端接口（简化对服务的编程式访问）也变得更加容易。

通常，REST 命令的输入和输出都是 JSON，这是一种人类可读的格式（尽管有些人可能会对此提出异议），非常适合用于探索性的工作，并且得到了通用编程语言的良好支持。

2. 代码接口实现

尽管 REST 是实现接口的一种常用且有效的方式，但有时候通过编程 API 实现更直接的客户端层也具有优势。以 Kafka 生产者为例，它可能涉及分区、缓冲、批处理和一些复杂的协议。虽然已经有了一个 Kafka REST 接口，Kafka API 仍然提供了更丰富、性能更高的访问接口。

你的目标应该是让接口变得更加通用。此外，开放性也很重要。如果要开发部署在产品中的代码，但这个产品不属于你，那么就要让产品团队能够访问实现细节。并不一定要访问源代码，但至少应包括文档、完整的接口定义等。此外，保持实现的开放性可以获得外部帮助，从而更好地清除 bug、优化性能和解决问题。

4.2.3 版本控制

版本控制很重要，特别是那些被各种应用程序频繁使用的接口。请注意，向后兼容性很重要，却令人头疼，需要进行额外的测试和规划。以下方法可以减轻它的副作用。

- 让迁移到新版本变得更容易。
- 不要频繁地更改 API。
- 主动分享已弃用的调用信息，以及特定版本支持和不支持的内容。
- 提供发布时间表。这将有助于内部团队确定交付主要功能的时间，并通过提供可预测的更新来促进接口消费者的工作。

Kafka 的版本控制经历了一个错误的阶段，不过后来回到了正轨。在 0.10 版本之前，如果生产者或消费者的版本与代理的版本不一致，就有可能发生故障。随着 0.10 版本的发布，客户端能够告诉 Kafka 代理自己所用的版本，然后代理就可以使用相应版本的协议与之通信。

除非在每次 API 发生变更时都进行全面的迁移（从规模上讲，这是不现实的），否则就应该预先设计一个强大的版本控制解决方案，可以从上述建议开始。

4.2.4 防御

虽然定义稳定且对用户友好的接口很重要，但是并非所有访问系统的用户都是友好的——可能是因为糟糕的应用程序实现、恶意行为或设计问题。

因此，必须考虑到其他人会如何利用接口来损害你的系统，并针对这些意外情况做好准备。以下是一些常见的状况。

- ❑ 倾斜

  如果系统采用基于分区的解决方案，那么倾斜会带来数不清的麻烦——某些分区会比其他分区大得多。找到发生倾斜的地方，并在造成灾难性结果之前制止它。请注意，数据倾斜超出了本书的讨论范围，但它的确是一个潜在的问题，在设计接口时需要注意。

- ❑ 负载

  无论是无意还是恶意的拒绝服务攻击，都有可能给系统带来不利的影响。采取措施找出负载峰值，并处理好它们。

- ❑ 奇怪的输入

  如果将输入格式设置为字符串，就必须假定传入系统的信息可以是任何内容。回想一下SQL 注入、空值或超大数值的问题，确保对系统的所有输入进行验证。

4.2.5 接口的文档和命名

在理想情况下，好的 API 文档应该简明扼要。如果你的 API 需要一本书来定义，那说明它太复杂。任何 API 或接口设计者的目标都应该是定义出不需要文档说明的接口。

文档应该涉及接口的行为方式、使用方式以及使用示例。

文档需要包含如下重要事项。

- 接口提供了哪些调用（函数）。
- 接口定义的调用参数，包括格式、schema、数据类型等。
- 调用输出，包括格式、schema、数据类型等。
- 状态需求。例如，服务需要维护状态吗？或是无状态的服务？
- 对并发场景的支持和并发场景中的行为。例如，支持多个请求。
- 已知的故障或可能的例外情况。

同样，函数的命名也应该简单明了。在理想情况下，函数的用途可以直接从函数名和参数中看出来。

尽量避免将技术解决方案放在函数名或定义中，尽可能将函数名定义为动词。接下来举一个现实世界中的例子，并将其与计算机世界联系起来。

示例是调用 `goToLocation(locationId: String)` 函数。在现实世界中，如果调用了 `goToLocation("Market")`，可以通过步行、开车、骑自行车等方式来实现，但不会让它们出现在函数名中。函数名需要体现的是函数正在执行的动作（动词）。

有一个笑话：命名是软件开发中最困难的事情之一。我们发现，减少命名时间的最佳做法是重用同一生态系统中其他优秀项目的命名风格。这样做有以下优势。

- 提供了可遵循的指南。
- 提供了客观的外部参考来支持决策。

最后，无论曾多么努力地设计，一两年后回顾时，都会希望自己当时能够做得更好。这就是这个世界的本质。以简单和可重复为目标可能是最好的选择。

4.3 非功能性考虑因素

前文提到，在设计接口时需要考虑一些次要的非功能性因素，包括系统的可用性、响应时间和负载容量。本节将探讨这些考虑因素。

4.3.1 可用性

所有接口都需要指定功能性合约，用于定义服务访问或与外部系统交互的接口还需要指定可用性合约。如果是为本地程序库定义合约，那么定义可用性就无关紧要了。但是，如果接口涉及外部服务，那么就要提供明确的服务可用时间和服务可用性保证级别。

可以将此视为给定服务的营业时间，以及为客户承诺营业时间的保证级别。在现实世界中，超市和影院都有规定的营业时间：假设超市是从早上 6 点营业到晚上 10 点，影院是从上午 10 点营业到午夜 12 点。

根据规定的营业时间，我们对超市和影院的开放时间有了一定的了解。不过，可能会发生一些事情扰乱预期的开放时间。例如，在暴风雪期间，影院可能会关门，而超市仍然会营业。在这种情况下，可以认为超市比影院具有更高的保证级别。

需要有两种可用性定义：一是服务何时可用，二是可用时间的保证级别。

你可能会问："在计算机世界里，服务不应该总是可用的吗？"来看一下不同可用性级别的示例。

- ❑ 计划维护时间窗口

  通常，服务会有一些不可用的计划时间段，用于对服务执行维护任务，例如升级软件、应用补丁和替换硬件。

- ❑ 只在特定时间段可用

  有些服务只需要在规定的时间段内可用，例如只可以在营业时间访问的服务。随着云技术的出现，以及按下按钮就能启动和停止服务的技术出现，只在特定时间段内提供服务可用性的做法变得非常实用和经济。

除了这些预期的可用性时间窗口之外，还有很多因素可能会破坏可用性，比如硬件故障、网络故障和云计算中断等。因此，在这些可用性窗口期间不可能承诺 100%的正常运行时间，不过有一些方法可以提高正常运行时间百分比，包括以下方面。

- 使用弹性组件。本书讨论的大多数作为大数据应用程序组件的系统，是为实现高可用性和抗故障性而设计的。例如，Kafka 提供了一种可扩展、可复制和高可用的数据集成解决方案。如果部署得当，Kafka 可以提供一个高弹性数据集成层。同样，大数据生态系统中的其他系统也可以在数据架构的其他层（例如存储层和处理层）中提供弹性。
- 为系统提供冗余。即使是最具弹性的分布式系统，也会发生故障，因此在任务关键型系统中拥有备份系统十分重要。这是一个复杂的话题,而且实现冗余的机制也因系统而异。有关如何部署这些系统的说明和详细信息，请咨询相关的供应商或参阅项目文档，以确保具有必要的可用性级别。
- 使用测试，包括负载测试和故障测试。4.3.4 节将讨论这方面的更多内容。

4.3.2 响应时间

与可用性一样，响应时间也不可能是完美的。系统故障、垃圾回收、网络延迟等因素都有可能影响服务的响应时间。

我们的目标应该是为用户提供一系列基于时间百分比的保证，它们应该经过验证和确认，如下所示：

- 95%的时间里响应时间为 10 毫秒；
- 99%的时间里响应时间为 50 毫秒；
- 99.99%的时间里响应时间为 1000 毫秒。

使用接口的一个好处是可以在不影响系统客户端的情况下提高可用性和减少响应时间。不过，拥有一个测试框架也非常重要，在每次发布时对系统进行负载测试，这有助于确保不违反可用性合约。

4.3.3 负载容量

负载容量定义了在给定时间段内可以处理多少请求，以及扩展的限制。同样，我们不会在合约中讨论如何提供负载保证，但会承诺将提供的负载容量级别。与可用性和响应时间一样，我们需要向接口用户提供详细的承诺，请看下面的示例。

当每秒处理不到 100 000 个请求时：

- 95%的时间里响应时间为 10 毫秒；
- 99%的时间里响应时间为 50 毫秒；
- 99.99%的时间里响应时间为 1000 毫秒。

当每秒有 100 000~200 000 个请求时：

- 95%的时间里响应时间为 20 毫秒；
- 99%的时间里响应时间为 100 毫秒；
- 99.99%的时间里响应时间为 2000 毫秒；
- 建议不超过每秒 200 000 个请求。

建议尽量让这些负载定义简单一些，并在适当的时候指出特定的限制。与可用性和响应时间一样，必须对每个发布版本进行负载测试，并密切监控生产系统的活动。

4.3.4 使用测试来确定SLA

无论选择了怎样的架构和组件，真正能够确定可用性和响应时间保证的唯一方法是对系统进行测试。测试需要在目标部署环境中使用真实数据和预期负载执行。

如果在接口中声明了系统不受节点故障的影响，那么就应该在测试系统和生产系统中定期测试随机节点故障。如果对在生产系统中进行这类测试有所顾虑，那么说明对系统的恢复能力缺乏信心，需要更多地考虑系统的故障恢复能力。

如果不在系统中测试和模拟故障，你只能纸上谈兵。如果声称系统可以处理节点故障，那么就不应该在凌晨 3 点叫醒技术支持人员。这样的故障应该在意料之中，而且系统应该具有自我修复机制。

4.4 通用接口示例

前文已经讨论了什么是好的接口设计，接下来看一下在创建系统接口时通用的一些架构模式。

4.4.1 发布–订阅

我们要讨论的第一个模式是发布 – 订阅模式，如图 4-10 所示。在这个示例中，有将消息发布到中央消息传递系统（代理）的组件，以及订阅代理特定队列的组件。

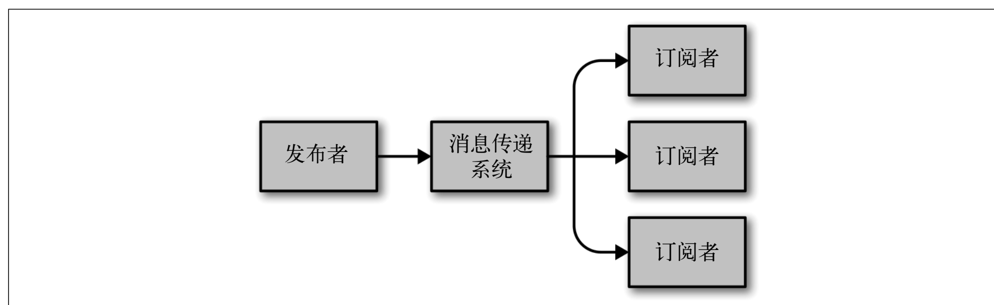


图 4-10：发布 – 订阅系统

该模式的中心思想是：发布者不需要关心与订阅者有关的任何事情。他们只需要关心如何发送消息，并保证消息的内容与接口所定义的内容相匹配即可。他们并不关心谁读取了这些消息或者读取者将如何处理这些消息。

在基于发布 – 订阅模式开发应用程序时，可以独立地开发各个组件。在某些情况下（如图 4-11 所示），订阅者的开发工作已经完成，但发布者的开发工作仍在进行当中。这时，可以创建一个模拟发布者，让开发工作和测试工作继续进行，避免中断发布者的开发计划。

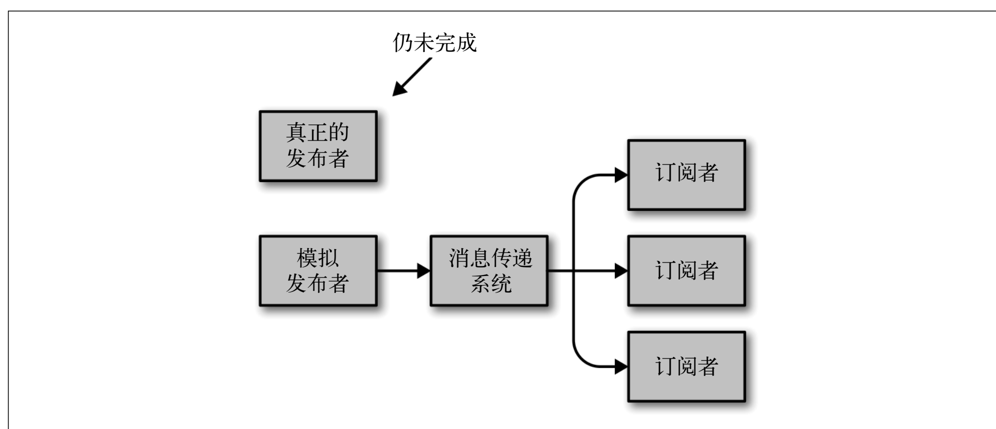


图 4-11：使用发布 – 订阅系统进行解耦

企业服务总线

企业服务总线（enterprise service bus，ESB）这种架构与发布 – 订阅模式有些类似，但规模更大。ESB 架构通常具有以下特性。

- 传输层
  需要可靠且可扩展的管道来接收消息，并将它们传递到一个或多个目的地。

- 发布者
  将事件发布到消息总线的系统。

- 消费者
  监听和消费消息的系统。

- 工作流
  包含了工作流逻辑的系统。可以接收事件和更新状态，确定需要做出怎样的动作，以及是否需要触发其他事件。

- 异步请求
  还可以使用异步请求模式。4.4.2 节将介绍这种模式。

❑ 仪表盘

仪表盘系统用于监控 ESB 的状态以及所有向它发布消息的系统的状态。

刚进入 21 世纪时，ESB 很流行。然而，在组织内部实现 ESB 时，在规模和协调方面存在一些问题。直到 Kafka 和集中式 schema 存储库的出现，ESB 概念才卷土重来。

4.4.2 异步请求–响应

我们已经讨论了发布 – 订阅模式，它是一种单向接口，但如果需要对请求做出响应，那该怎么办？这个时候可以使用请求 – 响应接口。接下来先介绍异步模式，稍后再讨论同步模式。

在查看示例之前，先来看一下什么时候需要考虑使用异步请求 – 响应模式。

- 想从组件中请求一些信息。
- 对于何时返回响应没有严格的需求。
- 可以接受重复的响应。
- 很了解架构的资源需求，例如用于持久存储请求和消息队列的内存需求。

如果这符合你的用例，那么可以参考图 4-12 所示的架构示例。

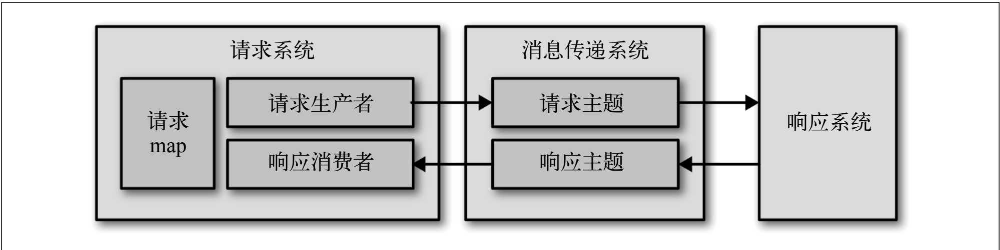


图 4-12：异步请求 – 响应架构

让我们来看看图中的一些组件。

❑ 请求系统

发出请求的系统。

❑ 请求 map

一种数据结构，用于存放已提交并正在等待响应的请求。因为存放的是未完成的请求，所以会占用内存。又因为内存是有限的，所以保存未完成请求的数量也是有限的。

❑ 请求生产者

将请求和键（用于索引请求 map）一起发送到消息传递系统的组件。

- ❑ 请求主题

  用于发送请求的专用队列。这个主题可以共享给其他请求，但最好在主题的使用与吞吐量和响应时间之间保持平衡。此外，同一主题中的请求应该被发送到相同的响应系统。

- ❑ 响应系统

  一个接收请求并返回响应的系统。理想情况下，这个系统本身没有状态，但可以与有状态的系统协作。

- ❑ 响应主题

  用于响应消息的专用队列。需要注意分配给响应主题的请求系统的数量。其数量越多，被请求系统丢弃的响应就越多，因为它们不需要那些发给其他请求系统的响应。

- ❑ 响应消费者

  监听响应主题的消费者，使用请求 map 将响应链接回发出请求的原始线程。如果该请求不在请求 map 中，说明响应已经被处理，或者响应针对的是另一个请求系统，只是使用了相同的响应主题。

这种接口设计的最大优点是可以解耦请求的处理与响应，并在时间限制方面为响应者提供灵活性，以便将结果返回给请求者。

对于需要花费很长时间（几秒到几小时）才能处理完的请求，这通常是一个很好的解决方案。但这并不意味着不能将它用在更加实时的解决方案中。假设使用 Kafka 作为队列，并在返回响应之前做一些基本处理，在这种情况下，可能会有几毫秒到几十毫秒的往返延迟，具体取决于所查询的系统。

4.4.3 同步请求–响应

我们要探讨的最后一种接口是同步请求 – 响应，这种接口对响应时间有严格的 SLA 要求。在这个模式中，不需要使用队列。相反，只需要老式的 Web 服务器。如图 4-13 所示，有一个请求系统发送请求，然后等待响应系统返回响应；请求和响应发生在单个事务中，这在响应保证、减少延迟以及消除重复响应方面具有优势。

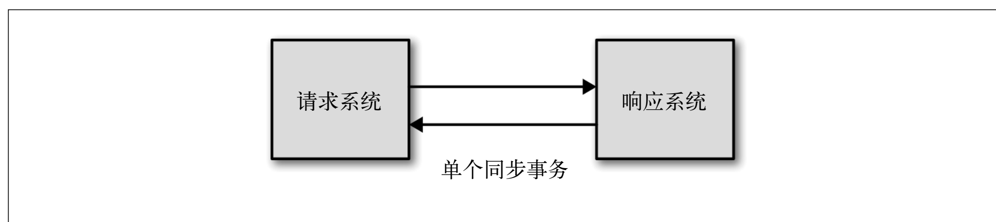


图 4-13：同步请求 – 响应架构

这可能会带来扩展性方面的问题，因为网络带宽、硬件资源等因素会影响响应时间和可处理的请求数量。另外，在应答系统不可用时，也面临一系列的挑战。如果担心 Web 服务器发生停机或过载，可以使用多个系统，并把这些系统部署在具有虚拟 IP 地址的负载均衡器后面，如图 4-14 所示。很多 Web 应用程序基本上采用这种架构。

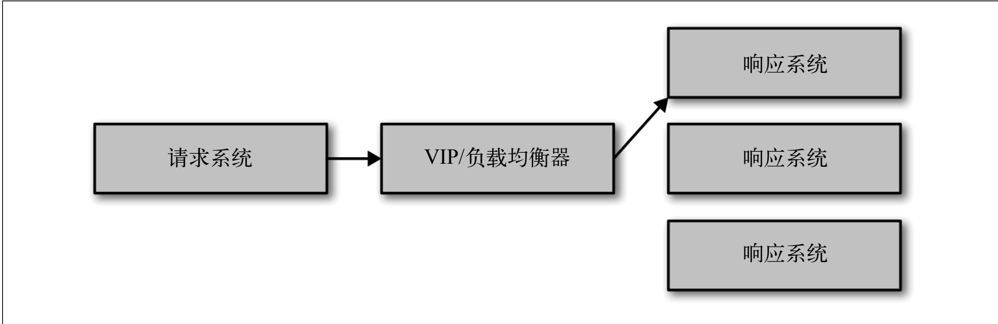


图 4-14：扩展同步请求 – 响应架构

4.5 小结

就像人体一样，可扩展和可维护的软件架构依赖于接口和抽象。设计良好的抽象有助于将具体的实现与整体架构解耦。这种解耦机制简化了开发过程，因为可以为服务创建模拟实现，并且不受系统内部实现变更的影响。

我们可以通过多种方式实现接口，例如标准 API、REST 以及发布 – 订阅系统。设计接口的选择会受到架构和需求的影响。如果系统是一个简单的 Java 应用程序，那么定义 Java API 可能就足够了。如果需要支持多种语言或从外部主机访问系统，那么可以考虑使用 REST 接口。

除了功能性接口之外，还需要注意非功能性需求，包括定义服务的可用性、响应时间和负载容量。

最后，在创建系统接口时，可以考虑几种架构模式，包括发布 – 订阅模式，以及异步和同步的请求 – 响应模式。

第 5 章

分布式存储系统

第 4 章讨论了接口设计。本章将讨论数据架构中可能会用到的分布式存储系统。在讨论各种可用的存储系统的同时，希望你能够渐渐理解为什么接口对于设计数据应用程序如此重要。

首先，本章会讨论分布式存储系统的一些核心属性，并对它们进行分类。然后，将深入研究一些当前广泛使用的分布式存储系统。我们将从讨论这些属性开始，因为本书不可能涵盖所有的存储系统，而且在本书出版之后，会涌现更多新的存储系统。所幸的是，在深入理解分布式存储系统的基础知识之后，你应该能够对新出现的系统很好地进行分类和评估。

5.1　分布式存储系统的属性

人们通过多种方式对分布式存储系统进行分类——有一些很有用，有一些则令人感到困惑。本节的目的是指出在为数据应用程序评估存储系统时有哪些重要的考虑因素。

请注意，第 2 章讨论了评估系统的注意事项，这与评估存储系统的注意事项有一定的关系。我们不打算在这里重申这些注意事项，但牢记它们有助于理解本章。

首先讨论当前主要的分布式存储系统的历史起源（剧透：其中大部分源于谷歌项目），然后再讨论以下可用来评估和分类存储系统的标准。

- 分区

  系统如何管理跨节点的数据分布？

- 可变性

  系统支持哪些修改数据的方式？

- ❑ 读取路径

  如何访问系统中的数据？

- ❑ 可用性和一致性

  在系统可用性和数据一致性方面，系统做出了哪些权衡？

- ❑ 用例

  系统解决了哪些问题？

5.1.1 谱系

在讨论存储系统时不一定要提到谱系，但考虑到分布式存储系统的复杂性，而且它们通常是基于现有系统的设计原则而构建的，所以还是有必要提及它。这里讨论的大多数系统都源于谷歌项目，比如谷歌文件系统（GFS）、BigTable、Spanner 和谷歌搜索。

举两个具体的例子：Cassandra 和 HBase 是基于 BigTable 构建的。这个谱系会告诉我们很多有关这两个系统的信息，包括以下方面：

- 它们的架构组件很相似；
- 它们是键 – 值存储系统；
- 它们具有很好的可扩展性；
- 它们将排序作为在磁盘上存储和索引数据的一部分；
- 本质上，数据是不可变的，但有些模式允许修改数据；
- 它们旨在解决相似的问题（例如，快速访问数据记录），并具有相似的性能特征。

图 5-1 是当前的分布式存储系统的谱系图。各个存储系统之间的关系可能存在争议，但该图仍然是探讨分布式存储系统谱系的一个很好的起点。另外，连接线并不代表下方的项目就一定是上方项目的扩展，它只是表明父项目对子项目产生过重大的影响。

可以通过以下方式来看待这些项目之间的连接关系。

- ❑ 谷歌项目的直接后裔

  其中一个例子是从 GFS 到 Hadoop 分布式文件系统（HDFS）的连接。这个连接表示开源社区创建了一个基于谷歌闭源解决方案的开源项目。

- ❑ 受启发的项目

  其中一个例子是从 Parquet 到 Kudu 的连接。Kudu 最初的想法来自 Parquet 列式文件，这种文件类型具有一些优势，例如更高的压缩率和更快的读取速度。

- ❑ 填补空白

  其中一个例子是从 HDFS 到对象存储的连接。HDFS 是一个糟糕的对象存储系统，所以后来出现了可以更好地支持对象存储的数据存储系统。

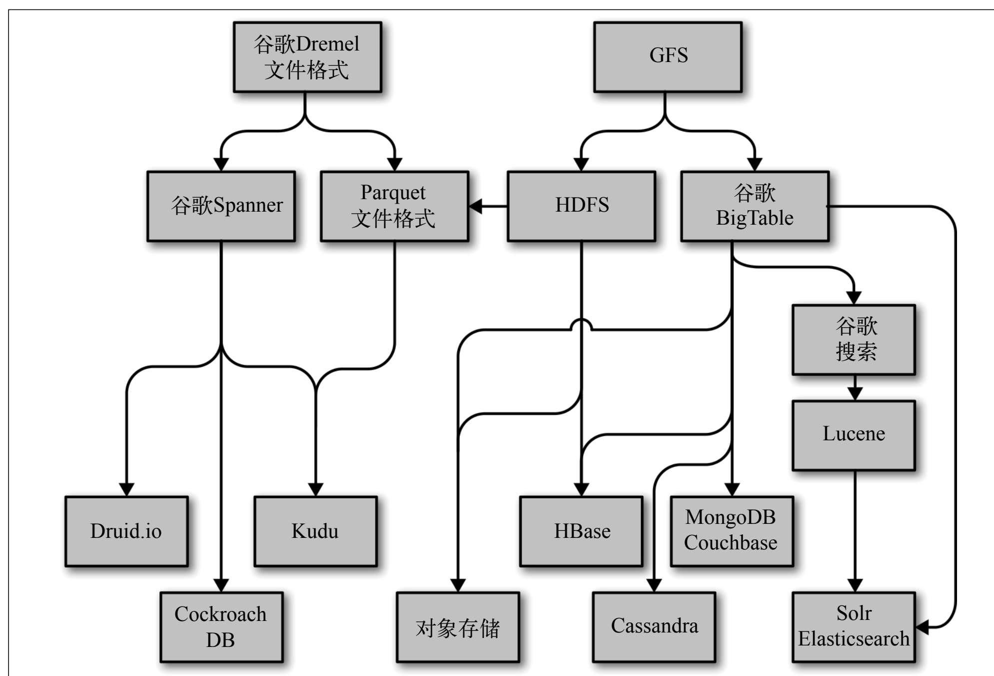


图 5-1：分布式存储系统谱系

5.1.2　分区

因为分布式存储系统使用了**分区**，所以具有很好的可扩展性——分区指系统如何确定要在哪些节点上存储数据，换句话说，它就是一种在整个系统中分布数据的机制。通常，分布式存储系统的分区类型是有限的，仅有**集中式分区**、**区间分区**和**散列分区**。下面将逐一介绍这些分区类型。

1. 集中式分区

GFS 和 HDFS 等存储系统使用了集中式分区，该分区类型依赖单个服务决定数据将驻留在哪个节点上。这种分区类型有一些优点，但缺点也很明显，因此它用得越来越少。

这类系统的优点是，即使数据节点发生故障，中央节点也可以确保数据均匀分布。中央服务知道所有节点的位置，并且可以在任何时候将数据分派给这些节点。

这类系统的缺点是存在潜在的瓶颈，因为只有一个元数据服务，所以会受到内存和多租户环境中请求吞吐量的限制。

此外，集中式分区系统会借助一种算法，该算法根据系统的当前状态在节点上放置数据。当我们转向其他类型的分区时，会发现它们没有将系统的当前状态作为分区的决策因素。这是因为大型分布式系统的状态维护成本很高，而这种成本会影响到性能和可扩展性。

2. 区间分区

区间分区是另一种常见的分区机制，它已经被用在 HBase 和 Cassandra 等系统中。但是，Cassandra 默认情况下会为键添加一个散列值。

字典（老式的硬复制类型）是区间分区的一个很好的例子。在字典中，字母表中的每个字母都有一个对应的区间分区。只要看一下字典就应该知道，区间分区策略存在一个根本性问题——**倾斜**。

当一个分区的内容明显多于其他分区时，就会出现倾斜。在字典中，字母 S 和 T 对应的单词比 X 和 Z 对应的要多得多。这对于字典来说不算太糟糕，但对于分布式系统来说，这会导致某些节点比其他节点做的工作更多。可以通过**散列**来解决这个问题。

3. 散列分区

在计算机科学领域，散列函数通常用来将给定的值映射到某个有限区间。在 Java 中，字符串 `ted` 和 `jon` 的散列值分别是 `114707` 和 `105417`。接下来，看看这个看似随机可重复的数字如何帮助我们将不同的记录放进不同的分区中。

```
val recordKey = "ted"

val hashCode = recordKey.hashCode
//hashCode = 114707

val absHash = Math.abs(hashCode)
//需要使用绝对值，因为散列值可能为负
//absHash = 114707

val numOfPartitions = 10
//分区数量根据数据表来定义

//找到散列的模和分区数量
val destinationPartition = absHash % numOfPartitions
//destinationPartition = 7
```

散列与倾斜问题有什么关系呢？有效的散列函数（以及巧妙地使用散列键）可以在分区之间均匀地分布数据，确保数据均匀地分布在集群中。

某些系统（如 Cassandra 或 Elasticsearch）使用了散列分区，但通常使用区间分区的系统也可以使用散列分区，方法是在键前面添加一个散列值。

5.1.3　处理数据变更

除了分区策略之外，分布式存储系统的另一个重要考虑因素是如何处理数据变更。当我们提及数据变更时，实际上说的是如何处理存储在系统中的记录的变更。常见的分布式存储系统针对特定类型的数据和访问模式进行了高度优化，并提供了不同的方式来处理数据变更。接下来将深入探讨存储系统中不同类型的数据变更模式。

1. 仅追加

一个需要考虑的因素是能否在原地修改数据。某些存储系统仅允许追加数据，就像将新日志行追加到日志中或者将新文件添加到文件夹中一样。在写入新日志或添加新文件后，就无法修改日志或文件中的值了。以 HDFS 为例，你可以删除和重写文件，但无法修改文件中的内容。

你可能会问：为什么要构建这种支持仅追加模式的存储系统？这是因为，在原地修改数据并不是一件容易的事情。你要么有固定的数据结构，而这些数据结构可能具有较低的压缩比，要么需要重写很多数据块。因此，在存储大量各种各样的数据时，固定的数据结构会造成资源浪费。此外，对于低延迟的存储系统来说，重写大型数据集的开销会很大。稍后将讨论一种更好的追加和压缩方法，所有基于不可变文件的 NoSQL 系统都采用这种方法来模拟数据变更。

2. 文件与记录

另一个需要考虑的因素是发生变更的级别——是文件级别还是记录级别。你可能已经猜到了，在基于文件的系统或对象存储系统中，数据变更发生在文件级别；而在其他存储系统中（例如 NoSQL 系统），数据变更发生在记录级别或文档级别。

通常情况下，知道了数据变更所在的级别，就知道了系统中事务的默认级别。例如，在文件系统或对象存储系统中，事务的级别为文件或对象，而在 NoSQL 系统中，事务的级别为记录。

3. 记录大小

还有一个需要考虑的因素是存储系统的最佳记录大小。某些系统（如对象存储系统和 HDFS）针对较大的数据块进行了优化，文件大小可以在 100MB 和 1GB 之间。而有些系统在数据块小于或约为 10KB 的时候表现得更好，例如 HBase 和 Cassandra。

数据块的大小与系统存储和修改磁盘上的数据有关。关键是要记住这些建议，从而充分地使用系统。选择不恰当的文件大小或记录大小可能会出现问题，例如，在使用 AWS Simple Storage Service（Amazon S3）存储大量小文件时就要特别注意。

与 HDFS 不同，对象存储系统不存在集中管理元数据方面的限制（稍后会详细介绍），所以它非常适合用于存储大量的小文件。但是，这可能会导致其他问题。假设使用 1KB 的文件来保存 10GB 的数据，就会有 1000 多万个文件。若在这个文件夹或表上运行 Hive 作业，那么在漫长的等待后，将遭遇内存溢出。Hive（和大多数执行系统）需要制定执行计划。执行计划需要确定读取哪些文件来执行查询，而 1000 万个文件已经超出了 Hive 的默认内存设置。

在这个示例中，使用 100MB~250MB 的文件会获得更好的执行性能。此外，由于每个文件包含了更多的数据，出现重复数据模式的可能性更大，因此很有可能会获得更高的压缩

比。用于处理大量文件所需的处理程序的数量也可能会影响性能。

因此，即使文件系统或对象存储不存在上述的小文件问题，仍然应该使用 100MB 或更大的文件。

4. 数据变更延迟

不同的存储系统针对具有不同事务边界的读取和写入批次大小进行了优化。例如，HBase 和 Cassandra 能以毫秒级的速度处理记录批次，换句话说，这些系统能够处理以毫秒级频率到达的记录批次。基于 Lucene 的系统（如 Solr）经过优化，可以以秒级或分级的速度处理记录批次，HDFS 则能够以分级的速度处理记录批次。

5.1.4 读取路径

在过去，只有单节点的关系数据库，它们的读取路径很简单。但现在的数据量很大，无法放入单个节点的内存中，需要分布在多个节点上，而不同的系统又提供了不同的数据访问方法。以下是分布式存储系统访问数据的方法。

1. 索引

索引被广泛地应用在分布式系统中，但通常在使用方式上存在差异。某些系统只索引文件的开头部分，而文件的所有内容都需要进行扫描过滤。还有一些系统会索引记录的每个字段，甚至是记录的实际内容。

通常，分布式系统中有 4 类索引。

- 文件级别的索引

  每个数据块一个索引。

- 记录级别的索引

  每个记录或文档都有一个主键，并使用主键的值作为索引。

- 简单的二级索引

  非主键字段的二级索引。

- 倒排索引

  基于 Lucene 的索引，所有内容都可以被索引，主要用于搜索和分面。

基于 Lucene 的倒排索引

如果对 Lucene 等系统和基于 Lucene 的解决方案（如 Solr 和 Elasticsearch）已经很熟悉了，可以跳过这部分内容。这部分将深入介绍倒排索引的工作原理。图 5-2 是一个针对字段 `color` 和 `size` 使用倒排索引的示例。

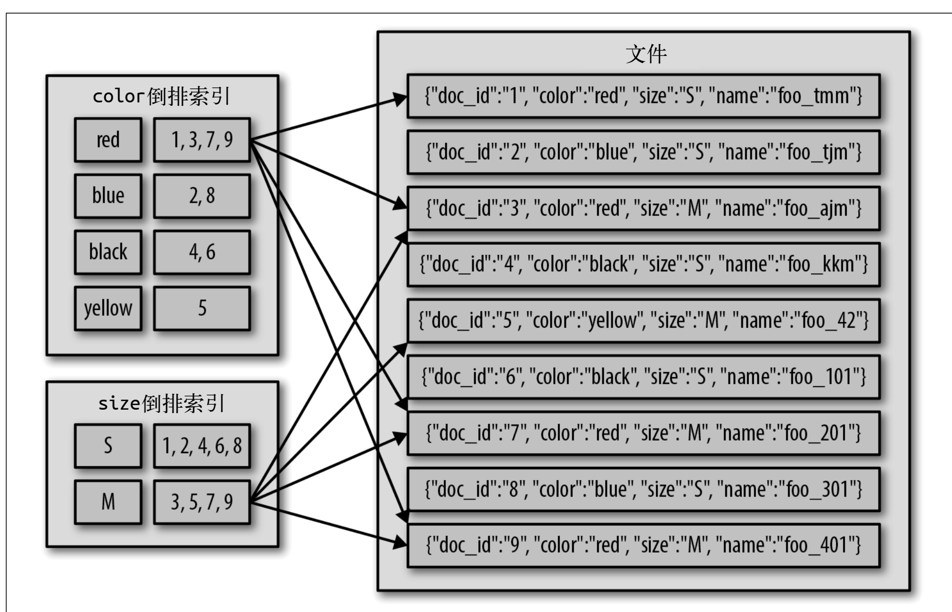


图 5-2：倒排索引示例

倒排索引与标准索引的最大区别在于，不使用 doc_id 来获取记录，相反，可以使用 "black" 找到两条记录，它们的 color 字段的值均为 "black"。

这在搜索记录时非常有用，也同样适用于根据不同字段值的计数制作图表。如果想计算包含 red、blue、black 和 yellow 的文档的数量，只需要计算 color 倒排索引中 doc_id 的数量。因此，可以不读取原始数据就生成图表。

此外，如果想根据 color 和 size 制作图表，可以通过合并连接来获得组合图表。图 5-3 显示了 red 和 M 之间的连接是如何发生的。请注意，由于数据已经经过排序，因此只需要边扫描 doc_id 边进行连接。

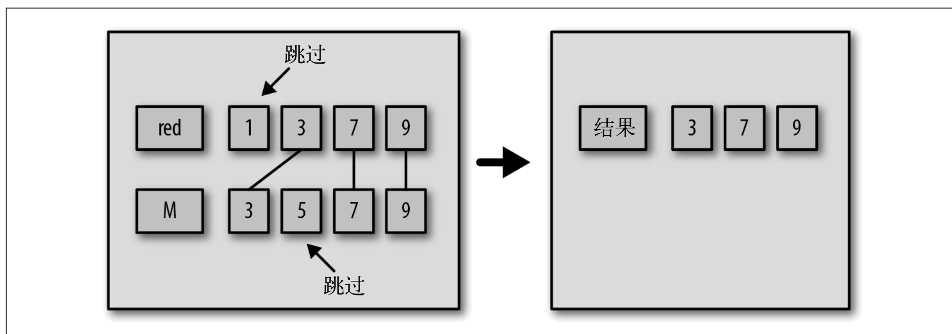


图 5-3：基于 Lucene 的合并连接

2. 行式存储与列式存储

“列式”（columnar）这个词可能有点令人困惑，因为有一些分布式存储系统（例如 HBase）也被称为“面向列的”系统。在本节中，当说到列式存储与行式存储时，指的是在存储系统中存储数据集的格式。为了更好地说明这一点，假设有一个包含多条记录的典型数据集，其中每条记录包含一组固定的列，并且每列都有特定的数据类型。图 5-4 展示了包含行和列的示例数据集。

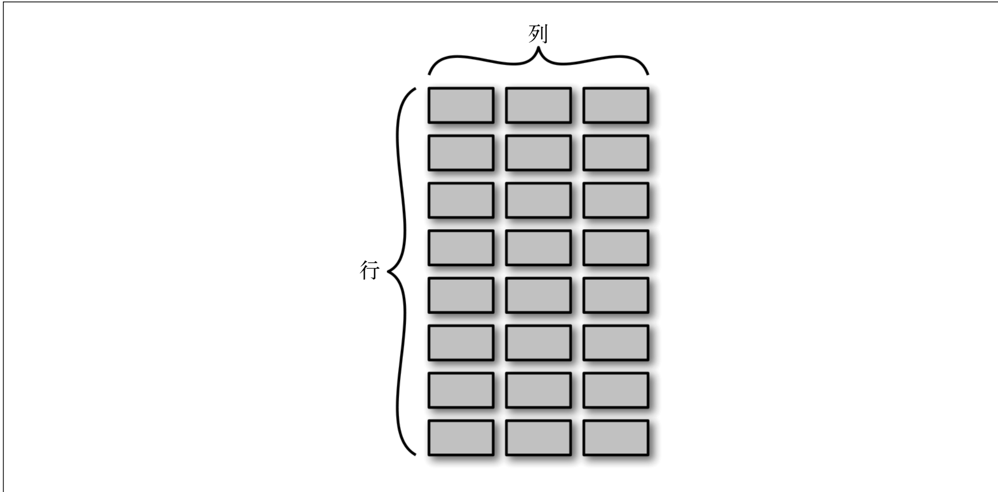


图 5-4：行式存储与列式存储

基于行的系统通过分组的方式保存和压缩数据，如图 5-5 所示。

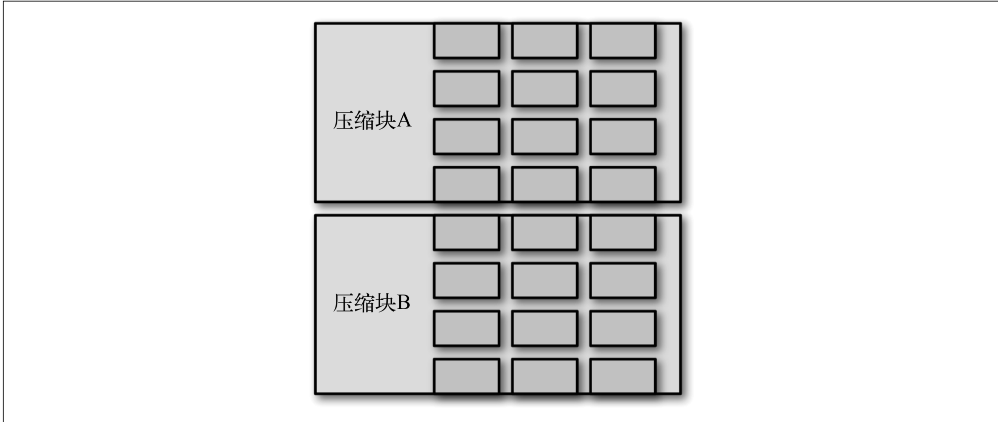


图 5-5：行式存储

在行式存储系统中，不同列的单元格是相邻存储的。这种存储方法可以减少写入数据和读取数据所需的内存。而在列式存储系统中，在写入文件时，每列都需要内存缓冲区来写入

和读取数据。因此，行式存储系统所使用的文件在写入和读取方面通常更高效。但是，速度还取决于其他大量的细节，具体请看下面的解释。

速度的比较

讲到这里，我们仍然很难确定基于行的格式比基于列的格式更快。先来看看为什么基于行的格式即使在读取数据方面的开销更小，也并不总是更快。

- 压缩

  就压缩率而言，基于行的格式通常不如基于列的格式高，因此，在某些情况下，吞吐量成了一个限制性因素（例如在使用 Amazon S3 存储数据时）。

- 代码路径

  基于行的格式有更简单的代码路径，但多年来，基于列的格式（如 Parquet 和 ORC）也经历了重大调优。因此，虽然列式的代码路径更复杂一些，但速度可能更快，因为它们优化得更好。

与往常一样，请使用你的数据对存储系统进行基准测试。

行式模型有两个缺点。第一个是压缩率，因为按不同列进行分组的单元格通常具有不同的数据类型，这不利于压缩——将相似的值分组在一起更有利于压缩。假设有一个包含股票代码记录的表，它由 3 列组成：股票代码、时间、价格。时间字段的值与其他时间字段的值（而不是股票代码字段的值）更相似。如果只想读取这些字段值的子集（一种常见的查询模式），就会发现第二个缺点——在行式模型中，即使只需要列的子集，也仍然需要读取所有列。

列式存储格式（如图 5-6 所示）有助于解决这些问题。我们将列存储在一起，这样可以更好地压缩数据，而且在查询字段值的子集时速度也更快。

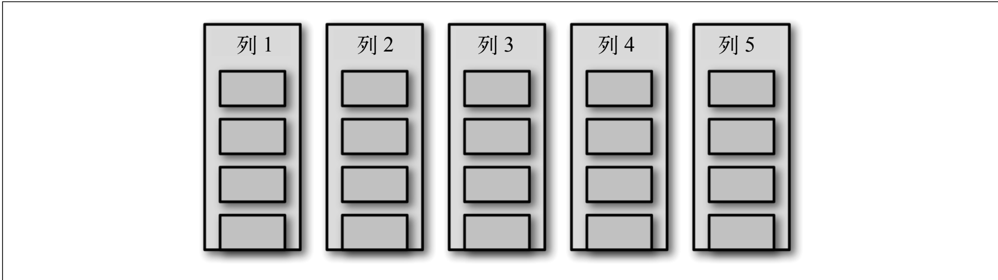


图 5-6：列式存储

3. 分区

在这里，分区有另外一种意思：将一个或多个数据字段作为键，然后基于这些键对存储数据进行分区。Apache Hive 等系统就支持这种分区方式，根据特定的列对数据表进行分区。

在使用这种分区时，可以在查询期间剔除不相关的记录，从而提高访问效率。如果记录是按照年份分区的，那么针对特定年份的查询就可以忽略与目标年份不相关的分区。

5.1.5 可用性与一致性

分布式系统不可避免地会出现故障，但系统的某些特征会影响它们的故障行为。因此，在发生故障时需要有所取舍，这取决于选择的存储系统。在讨论这些权衡之前，先简要介绍一下 CAP 定理。

1. CAP 定理

根据维基百科的描述，CAP 定理表明，分布式数据存储系统不可能同时提供以下保证中的两种以上。

- 一致性保证：每次读取都应该得到最新版本的数据。
- 可用性：每次读取都会得到有效的响应，但不保证是最新版本的数据。
- 分区容错：即使节点间出现数据丢失，系统仍然可以继续运行。

由于网络故障在分布式系统中是不可避免的，因此我们必须容忍网络分区，这意味着需要在一致性和可用性之间做出选择，如图 5-7 所示。

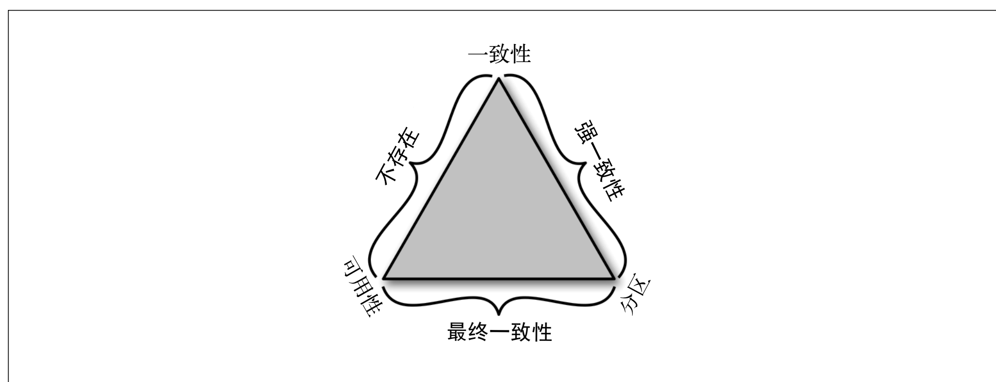


图 5-7：可用性与一致性

2. 选择可用性：最终一致性

最终一致性是指所有的数据请求最终将返回相同的值。这个模型为了获得更高的可用性而放弃了数据一致性。此时，如果系统出现故障，可能会出现数据丢失。而且更重要的是，我们无法保证在请求时会获得最新版本的数据。

3. 选择一致性：强一致性

顾名思义，强一致性可以保证对特定数据的访问总是返回相同的值。因此，当为了一致性而放弃可用性时，如果系统出现故障，我们就无法在给定时间内读取和写入系统分区。

选择一致性还是可用性通常取决于实际情况。不过，有时候也会由选择的存储系统决定。例如，HBase 就选择了一致性，而 Cassandra 选择了可用性。不过，Cassandra 还是为一致性模型提供了一些控制机制。

5.1.6 主要用例

最后一个需要考虑的问题是存储系统的主要使用模式。我们可以在多个用例中使用一种存储系统，但有些系统针对一小组使用模式进行了高度优化，如果使用不当，就会导致失败。在介绍不同用例的过程中，将会说明哪些系统适用于哪些用例，并在介绍每个系统时提供更多的上下文信息。

1. 大型扫描

这类用例要求对数据进行大型扫描，例如按照日期或其他大型分区边界对数据进行分区，但分区中的记录不少于 100 万条。这类用例主要包括查询很大的记录块、机器学习、图处理和窗口操作。对于这些用例，通常会使用基于文件的存储系统（例如 HDFS）或对象存储系统。

2. 随机数据访问

这类用例要求快速访问一条或多条记录，或者在大约一毫秒内更新记录，而不是扫描很大的数据块。为此，需要一个高度可变和支持索引的系统。像 HBase 或 Cassandra 这样的 NoSQL 系统通常最适合这类用例。

3. 数据立方体

这类用例要求进行大量的深度分析。在这个模型中，需要对数据进行高度优化，以支持从多个角度查询数据。基于 Lucene 的系统（如 Elasticsearch 和 Solr）可以作为这类用例的解决方案。

4. 时间序列

在物联网应用程序中，需要存储和访问大量与时间相关的事件。为此，我们需要一个优化过排序的存储系统。对于这类应用程序，可以将 NoSQL 系统（如 HBase）和数据存储系统（如 Druid）作为存储解决方案。

5. 高可变性

这类用例要求以非常快的速度改变系统状态，例如排序列表或流式时间窗口。它们一般是基于内存的系统，例如 Druid 或 Redis，它们针对某种类型的状态变化进行了优化。

5.2 存储系统细分

到目前为止，我们已经讨论了以下几点。

- 谱系

  了解系统的起源。

- 分区

  了解系统在不同节点上存放数据的策略。

- 处理数据变更

  了解系统的写入行为以及如何处理数据变更。

- 读取路径

  了解系统的数据访问行为。

- 可用性与一致性

  了解系统对数据的可用性和一致性的权衡。

- 主要用例

  了解系统的设计目标。

接下来将基于这些考虑因素，介绍当前最为流行的开源分布式存储系统。

5.2.1 HDFS

先从 HDFS 开始，看看它具有的存储系统特征，以及这些特征是否符合我们的要求。

1. 谱系

HDFS 是 Apache Hadoop 项目的一部分，于 2006 年左右作为一个开源项目启动，并在早期得到了雅虎的支持。HDFS 本质上是 GFS 的一个开源移植版本。

Hadoop 最初用于开发互联网倒排索引框架，以支持雅虎搜索。这个应用程序需要存储大量的数据，并通过 Hadoop MapReduce 高效地处理分布式数据。

2. 分区

在 HDFS 中，有一个名称节点，用于保存集群数据块的位置元数据。名称节点负责指定和获取块节点分配，以及处理再均衡操作。

HDFS 采用集中式分区模式，所以具备这种模式的所有优缺点。

3. 处理数据变更

HDFS 只能通过追加的方式修改数据，你可以一直追加数据，直到文件关闭。关闭后的文件是不可变的。如果要修改 HDFS 中的数据，需要重写相应的文件。在重写文件时，通常会使用一种叫作**压缩**的模式。图 5-8 大致说明了压缩机制。

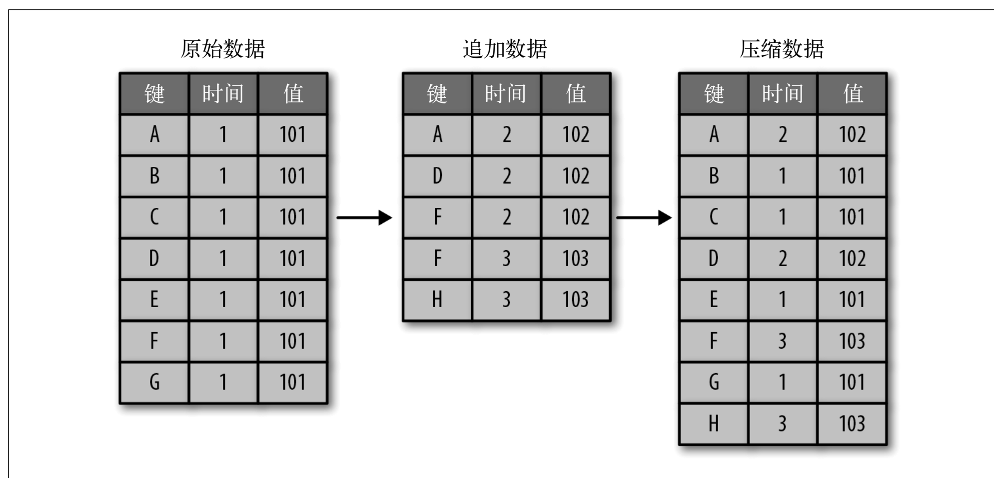


图 5-8：通过压缩修改 HDFS 文件

几乎在所有允许修改数据的系统中都能看到这种模式。但与大多数系统不同的是，HDFS 的压缩需要手动执行，而 Cassandra、HBase、Solr 等系统会在后台自动执行压缩。另外，与那些为修改数据而进行优化的系统不同，HDFS 在完成整个文件的重写之前，无法显示新修改的记录。

4. 读取路径

在很大程度上，HDFS 是为扫描大块数据而设计的。不过，HDFS 有一些高级的文件格式，例如 Parquet 和 ORC，这些格式让用户可以借助下推过滤器（pushdown filter）来避免读取行和列。随机访问记录实际上是不可能的，对索引的支持也非常有限。

5. 主要用例

如果在本地存储了大量数据（TB、PB 或更多），并且想执行需要访问大块数据的查询或处理，例如机器学习或 ETL，那么可以考虑使用 HDFS。

此外，因为 HDFS 位于数据中心，所以让存储节点附带计算能力是一种很正常的做法。HDFS 之所以能在很长一段时间内受到推崇，是因为可以直接在存储数据的节点上处理数据。

5.2.2　S3和对象存储系统

接下来介绍 Amazon S3 等对象存储系统。

1. 谱系

在使用 HDFS 时，关注的是数据本地性。然而，云存储的出现引发了数据本地性“大逃离”，这主要是因为云存储使单独为存储和处理付费变得切实可行，这为优化成本和资源

提供了很多选择。Hadoop 和 HDFS 鼓励将处理和存储放在一起。图 5-9 显示了云存储提供的一些选项。

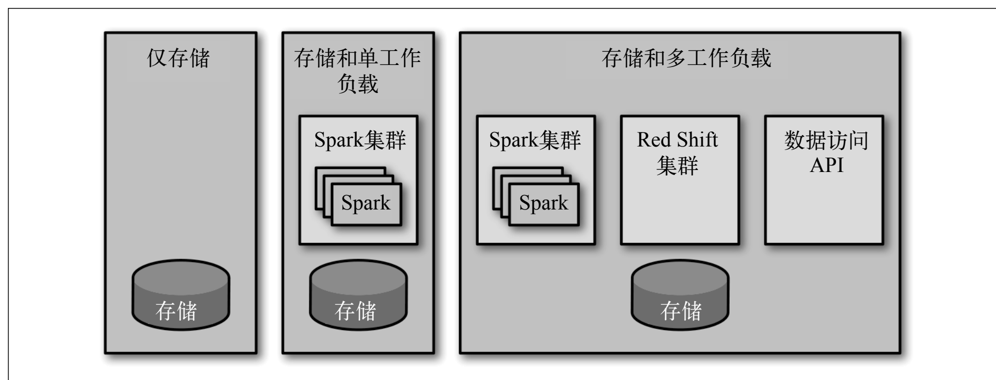


图 5-9：云存储选项

从本地性到云端非本地性的转变促成了**对象存储系统**的崛起。Amazon S3 是当前主要的案例。其他云服务提供商也提供了自己的对象存储系统，例如微软的 Azure Blob 存储和谷歌的云存储。即使是本地云解决方案，也提供了对象存储，例如 Swift。

对象存储通常会放弃本地性和快速元数据扫描，并采用元数据最终一致性模型。请注意，这里的一致性是指元数据的一致性，而不是实际数据的一致性。具体以 HDFS 为例，集群数据块的位置集中保存在名称节点上。名称节点将在写入所有数据块后记录每个数据块的位置。

相反，在 S3 这样的对象存储系统中，在写入数据或移除数据之后列出存储桶中的对象，可能不会立即返回正确的结果。虽然 S3 模型存在一些不足，但因为摈弃了集中式分区方案，所以为现代云数据存储提供了必要的规模。

2. 分区

具体的实现各不相同，但所有对象存储系统都使用某种形式的散列分区来分布数据。

3. 处理数据变更

对象存储系统与 HDFS 一样，以文件或 blob（binary large object，二进制大对象）作为写入单元，而且这些文件或对象只能通过追加的方式修改，在追加后是不可变的。此外，HDFS 所有的写入模式、文件类型和目录模式也都适用于对象存储系统。你甚至可以在对象存储系统上运行 Hive，在文件和文件夹之上创建表。

对象存储系统与 HDFS 等系统的主要区别在于，压缩数据在前者中有着更重要的地位。原因有二：首先，可以降低存储成本；其次，在处理数据时需要将数据传输到计算节点，压缩数据可以减少网络带宽的使用。

4. 读取路径

与分布式存储系统一样，对象存储系统也通过网络传输数据，因此在必要时需要进行压缩和缓存。如前所述，对象存储系统和 HDFS 等系统之间的一个区别在于元数据管理。从好的方面来说，对象存储系统不存在像 HDFS 那样的单个名称节点瓶颈问题。不好的方面在于，因为元数据具有最终一致性，所以有可能会过时，而且检索文件和目录列表的效率不如 HDFS 高。

5. 主要用例

如果在云端运行应用程序，可以考虑将对象存储作为长期的数据存储系统，替代 HDFS 等系统。即使没有在云端存储数据，也可以了解一下 OpenStack 的 Swift 等技术，并将它们用在数据中心里。

5.2.3 Apache HBase

在讨论了基于文件和对象的存储系统（如 HDFS 和 S3）之后，继续来讨论针对特定访问模式而设计的存储系统。先从 HBase 开始。

1. 谱系

2006 年，谷歌发布了一份有关 BigTable 的白皮书。BigTable 是一个运行在 GFS 上的 NoSQL 存储系统。HBase 是首批实现这种架构的开源项目之一。HBase 在 HDFS 之上运行，为 HDFS 环境提供了一个用来创建可变数据集（可以使用索引）的选项。

2. 分区

与其他 NoSQL 解决方案不同，HBase 会尽可能保留原始的键和值——键、列、单元格都是字节数组。这是一个强大的特性，但也带来了一些复杂性。

HBase 基于键的字节数组来定义分区区域，如果键设计得不好，就很容易出现数据倾斜。可以使用散列分区，但只能将散列值添加到字节数组键的开头。

3. 处理数据变更

在 HBase 环境中，客户端连接到 HBase 主服务器并获取分区。在知道有哪些分区之后，客户端就可以自由地向各个分区（区域）首领写入数据。与所有 NoSQL 解决方案一样，HBase 允许修改单行或批次数据。每个单元格对应底层 HFile 文件中的一行，因此修改单元格数据就有了可能。

4. 读取路径

HBase 是为进行快速读取和写入而设计的。通常，如果记录还没有被放进缓存中，从磁盘获取记录只需要进行一次查找。

为了优化读取路径，HBase 提供了两个内存缓存选项：memstore（缓存刚刚写入的数据）

和 blockcache（缓存最近获取的数据）。

对于每秒能够处理 30 000~100 000 条记录的服务器来说，所有的读取操作和写入操作都应该是毫秒级的。

5. 可用性与一致性

HBase 是强一致性的，所以如果有一个节点发生故障，那么给定分区（区域）将会出现一段时间的停机（通常在几秒内）。因此，HBase 不是实时应用程序（比如网站）的理想选择。Cassandra 可能会更合适，因为它提供了更灵活的一致性选项。

6. 主要用例

HBase 主要用于需要强一致性的 NoSQL 存储，比如用来保存股票交易关系的大型图数据库。

通常，HBase 适用于需要快速修改数据、快速获取数据、进行数据排序和动态创建列的应用程序。

此外，HBase 经常被拿来与 Cassandra 做对比（5.2.4 节将讨论 Cassandra 及其优点）。不过，HBase 在某些方面比 Cassandra 更具优势。

- 具有更高的节点存储密度。在某些生产环境中，单个 HBase 节点可以存储超过 30TB 的数据，而单个 Cassandra 节点最多只能存储 5TB 的数据。这主要是因为 HBase 在 HDFS 上运行，而 HDFS 最初的设计目标之一就是支持大节点容量。
- HBase 的批量写入操作更容易进行扩展。随着集群的增长，Cassandra 默认的批处理机制可能会出现问题，主要是因为 Cassandra 没有像 HBase 那样的分区首领。
- 故障恢复更加容易。在 Cassandra 集群中，如果一个节点发生故障，必须使用新节点替换故障节点（使用自动伸缩组可以很容易地完成这个操作）。而 HBase 只需要重新分区，然后继续运行。

5.2.4 Apache Cassandra

尽管 Cassandra 的目标用例与 HBase 的类似，但在设计中采用了不一样的方法。

1. 谱系

大约在发布 HBase 的同一时间，Facebook 正在开发自己的 NoSQL 存储系统——Cassandra。最初，两者之间最大的区别是一致性模型——HBase 采用了强一致性模型，Cassandra 则采用了可调整的一致性模型。

可调整的一致性

之前，我们讨论了 CAP 定理以及如何在强一致性和最终一致性之间做出选择。虽然这个定理仍然有效，但也可以改变它。为了解释清楚这个问题，来看看 Cassandra 如何通过不同的写入和读取配置来实现可调整的一致性。

一个节点写入，一个节点读取

在 Cassandra 中，可以定义一次读取或一次写入需要涉及多少个节点。最快的方法是每次写入和读取只涉及一个节点，如图 5-10 所示。

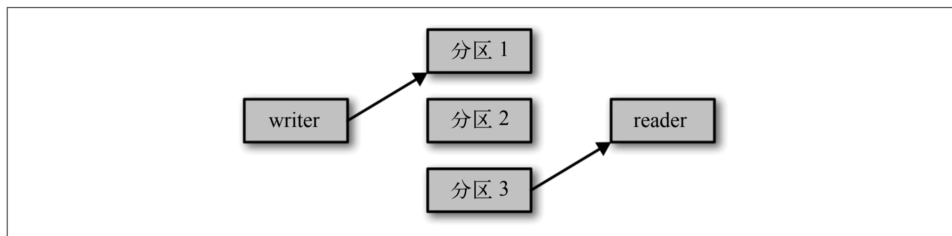


图 5-10：单节点读写一致性

图 5-10 表明，如果在写入之后立即读取，可能会得到旧的结果，因为写入的数据可能还没有复制到被读取的分区。

请注意，这个模型可以允许两个分区发生故障，并且不会影响读写行为，这是最终一致性的一个优点。另外，单节点写入模式发生数据丢失的可能性更高。如果写入的节点在复制数据或将数据保存到磁盘之前发生故障，就有可能会丢失数据。

仲裁读写

如果写入和读取的节点数量大于分区数量，就可以获得写后读一致性。由于读取和写入的节点比复制节点多，因此可以始终保证至少有一个读取操作与其中一个写入操作是重叠的。比如两次写入和两次读取，如图 5-11 所示。

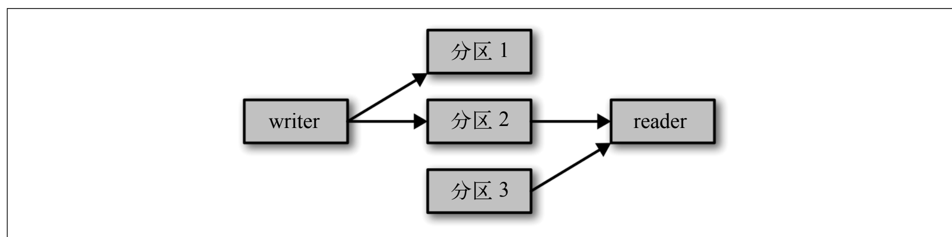


图 5-11：多节点读写一致性

从图 5-11 中可以看到，reader 将从分区 2 和分区 3 读取数据。分区 3 包含要读取的数据，但它们的时间戳可能比分区 2 的旧，所以 reader 将读取分区 2 的数据。

如果一个节点发生故障，仍然不会有什么问题，如图 5-12 所示。

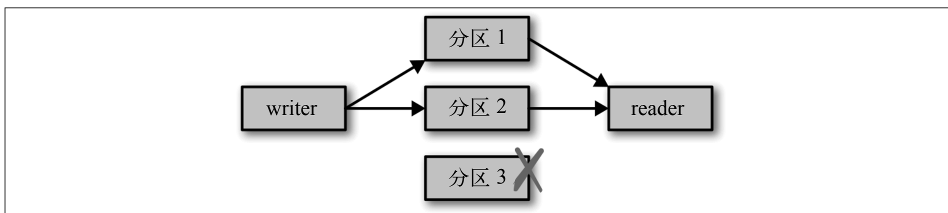


图 5-12：发生节点故障时的一致性

这是 CAP 定理的灰色地带。虽然遇到了故障，但并没有失去可用性。CAP 定理在这个时候仍然有效，因为 Cassandra 扩展了故障的定义。在仲裁配置中，故障是指有两个节点出现问题。

这种对 CAP 定理的扩展也是有代价的。与强一致性的系统相比，它的性能会差一些。为了更好地说明这个问题，图 5-13 比较了 HBase 和 Cassandra 的仲裁写入。

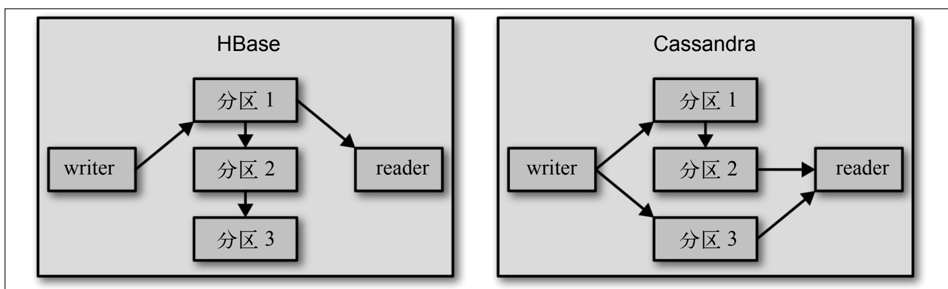


图 5-13：HBase 与 Cassandra 的一致性对比

请注意，HBase 中有一个首领节点，所以 writer 和 reader 始终只与一个节点通信，而 Cassandra 的仲裁配置需要两个节点。

其他选项

还有其他 Cassandra 配置选项。例如，可以写入 3 个节点，然后从一个节点读取，或者写入一个节点，然后从 3 个节点读取，如图 5-14 所示。这样就可以将性能重点放在其中的一方。

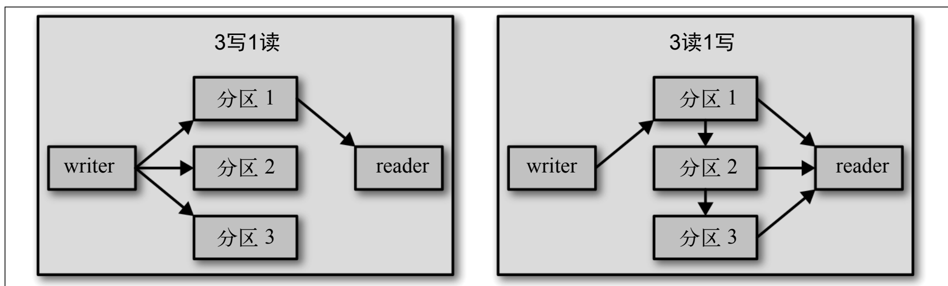


图 5-14：Cassandra 一致性——其他配置选项

此外，Cassandra 甚至可以将故障定义扩展到集群。你可以在集群之间建立仲裁，如果有 3 个集群，那么当其中一个发生故障时，不会带来任何影响。但是，这种配置存在额外的性能损失。

Cassandra 还提供了一种叫作 CQL 的查询语言。CQL 提供了一个 SQL 风格的接口，让用户能够更容易地操作 Cassandra 中的数据。此外，Cassandra 没有将 HDFS 作为底层数据存储，从而为部署提供了更大的灵活性。

2. 分区

与 HBase 一样，Cassandra 也采用了区间分区。但与 HBase 不同的是，Cassandra 不需要手动定义键，而是在后台自动进行散列。此外，在 Cassandra 中创建表的时候，可以借助 CQL 通过双键分组方案进行简单的手动分区和排序。

图 5-15 显示了在 Cassandra 中使用两个键创建数据表的示例。

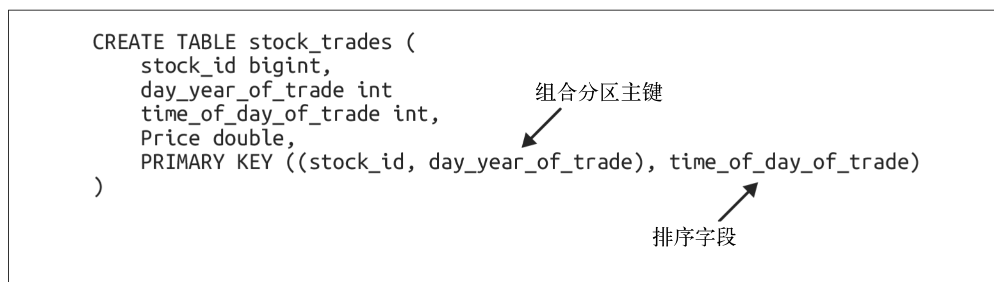


图 5-15：使用两个键创建 Cassandra 数据表

这将按照库存和日期（年、日）对记录进行分区。然后，分区中的所有价格都将按照时间进行排序。需要注意的是，与 HBase 等系统相比，在 Cassandra 中配置这种数据表更容易。

这种简单的分区可能是 HBase 和 Cassandra 之间最重要的区别。在 HBase 中，任何任务都需要编写代码来完成。而在 Cassandra 中，如果会用 SQL，工作效率就会非常高。但是，与其他接口一样，为了获得高性能，需要快速了解自己的 CQL。

3. 处理数据变更

在大多数情况下，Cassandra 的写入模式非常像 HBase，主要专注于快速的单记录更新。

4. 读取路径

在读取路径方面，Cassandra 仍然与 HBase 非常相似，但它更侧重于可调整的一致性。此外，Cassandra 可以从任意的副本读取数据。HBase 的写入和读取都发生在分区的首领节点上，Cassandra 的读取和写入则可以发生在任意一个副本节点。

5. 主要用例

适合使用 HBase 的场景通常也可以使用 Cassandra。是选择 Cassandra 还是 HBase，主要考虑一致性模型和故障处理的差异。

5.2.5 Elasticsearch和Apache Solr

接下来将讨论基于 Lucene 的系统，例如 Elasticsearch 和 Solr。

1. 谱系

Lucene 是一个用来索引 Web 内容的开源项目，可以实现 Web 搜索引擎。Elasticsearch 和 Solr 是构建在 Lucene 之上的两个查询引擎实现，提供了一些以 Lucene 的结果为基础的功能。

2. 分区

对于这些引擎来说，散列分区是非常关键的。这些引擎中的大多数查询操作是 reduce-by-key 查询，基于散列的分区机制为这些系统提供了良好的存储和可查询分区。因为每个文档都有一个 ID，并且所有的散列都是针对 ID 进行的，所以可以避免出现数据倾斜问题。

3. 处理数据变更

这些系统更新文档的能力在不同的系统和版本之间有所不同。一般来说，它们都支持更新文档，可能是更新整个文档或文档中的特定字段。但是，在底层可能需要删除旧文档，然后使用新文档替换。

请注意，由于在更新文档时需要存储不同的版本，因此这种数据变更模型会导致存储需求的增长。这也意味着在更新文档时，查询时间会有所增加。出于性能方面的考虑，如果你的需求包括数据的高可变性，那么基于 Lucene 的搜索工具可能不是最佳选择。

4. 读取路径

NoSQL 系统使用主键来查询记录，在某些情况下还可以使用辅助索引，而基于 Lucene 的系统可以通过任意索引字段查询文档。此外，Lucene 引擎不仅仅是为了获取记录，它其实是一个搜索引擎，所以最终返回的内容和顺序都非常重要。

此外，在某些情况下，我们可能不需要请求具体的数据记录，只需要根据索引计数生成图表。随着 Kibana（用于 Elasticsearch）、Banana（用于 Solr）和 Hue（支持多个系统）等动态图表工具的兴起，这类情况变得越来越普遍。它们降低了从这些系统中获取数据价值的门槛。

5. 主要用例

如果需要一个搜索引擎或实时的数据立方体，它们都是合适的工具。但请注意，所有的索引都会增加这些系统存储数据的成本。因此，尽管这些系统非常有用，却很难进行扩展。

5.2.6 新进者：Apache Kudu和CockroachDB

在开源的企业数据世界里，并不是每天都会有新想法出现并大获成功。本节将讨论两个相对较新的解决方案：Apache Kudu 和 CockroachDB。

1. Kudu

Kudu 是一个 Spanner 风格的系统，源于 Cloudera。它受到 Parquet 的最初承诺和列式文件格式的启发，看起来就像是 HBase 和 Parquet 的孩子。Kudu 具备了接近 HDFS 的扫描速度和 NoSQL 的可变性。不过，它在数据扫描方面不太可能比 HDFS 更快，也不会像 NoSQL 那样修改数据或访问单条记录。

它的底层存储使用了列式格式，这有助于数据扫描，但要以 PUT 和 GET 的性能为代价。因为列式格式需要在不同的文件或位置写入每一列——对于 Kudu 来说，需要写入到不同的文件。这意味着如果有 100 列，那么在使用 GET 获取一条记录时需要查找 100 次磁盘。如果在机械式硬盘上执行这样的操作，那么执行最基本的 PUT 和 GET 需要花费很长的时间。因此，在使用 Kudu 时，建议使用固态硬盘。

由于存在这些限制，Kudu 主要适用于需要几近实时修改数据的场景。本书不会具体地讨论 Kudu，但会提到适合扫描的数据表，比如 HDFS、S3 或 Kudu 的数据表。

2. CockroachDB

CockroachDB 这个名字可能是所有存储平台中最奇怪的一个。蟑螂（cockroach）很难被杀死，CockroachDB 就是借鉴了这个寓意。可以将 CockroachDB 视为 Impala 和 Kudu 的杂交，只不过它还提供了事务和嵌套类型。在 Kudu 中，SQL 层独立于存储系统，而 CockroachDB 更像是 Cassandra 或传统的关系数据库，它将 SQL 作为与系统交互的主要方式。

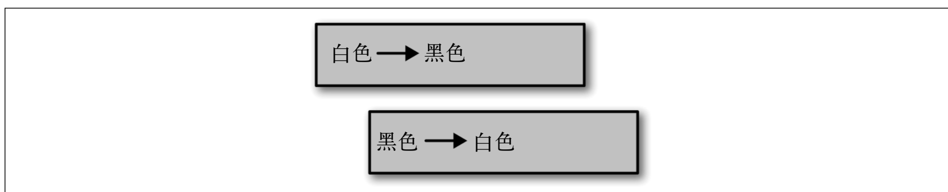
CockroachDB 之所以能够在所有 Spanner 开源解决方案中脱颖而出，是因为它提供了具备快照隔离和可串行化快照隔离的事务机制，这是维护数据完整性的重要机制。为了更好地理解不同事务类型之间的区别，下面提供了更多有关事务类型的详细信息。

快照隔离与可串行化快照隔离

尽管事务是传统数据管理系统（如关系数据库）的组成部分，但在开源的分布式数据管理系统中并不常见。因此，为了更好地说明事务隔离类型之间的差别，这里使用了来自数据库专家 Jim Gray 的一个示例。在这个示例中，一个袋子里装有 4 个球：2 个黑球和 2 个白球。然后，做出如下修改并提交。

- 将所有白球改为黑色。
- 将所有黑球改为白色。

以下是事务开始和结束的时间线：



在使用可串行化快照隔离时，这两个命令将逐个执行，如图 5-16 所示。

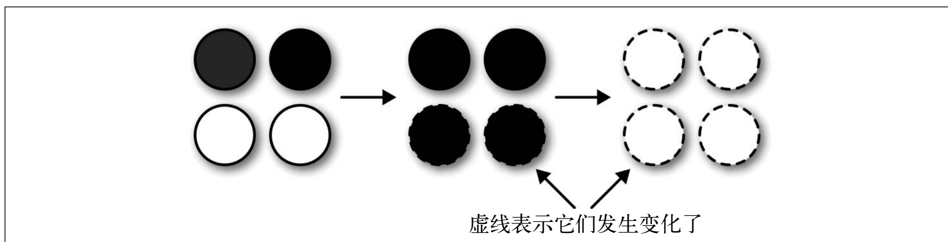


图 5-16：可串行化快照隔离事务

在使用快照隔离时，将看到不一样的结果。因为命令是并行触发的，它们会独立修改不同的球，如图 5-17 所示。

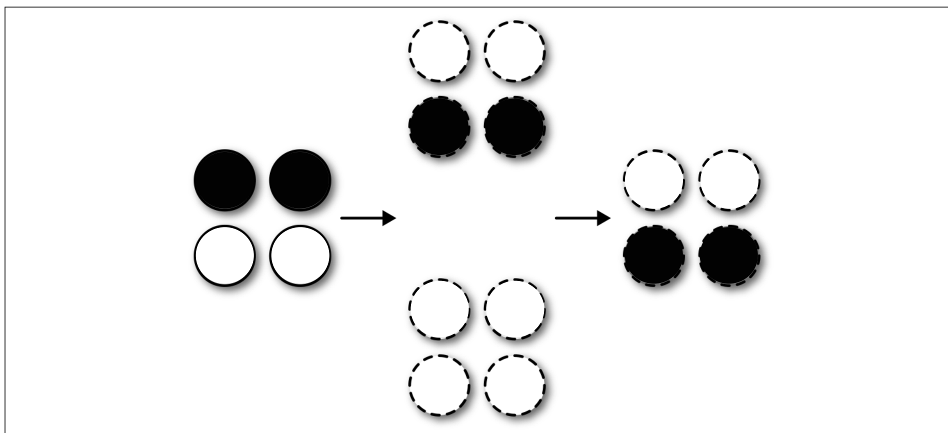


图 5-17：快照隔离事务

5.2.7 内存存储系统

本章要讨论的最后一种存储系统是内存存储系统。前文讨论过的大多数存储解决方案使用了某种类型的内存层进行缓存和（或）缓冲。此外，有很多常用的数据缓存系统可以作为应用层的一部分，例如 Memcached 或 Oracle Coherence。本节谈论的是长期的存储系统，它们将内存存储作为存储的核心部分。这是一个很大的类别，无法完全覆盖。本节只讨论

其中的两种，它们可以代表内存存储系统所能提供的特性。

- Druid，提供高性能的并行操作。
- Redis，支持高级数据结构。

Spark Streaming、Flink 和 Kafka Streams 等流式引擎也可以提供类似的功能，因为它们在引擎内提供了持久化状态。本章重点关注存储系统，第 8 章会用一定篇幅讨论流式引擎。

1. Druid

先从 Druid 开始，它与 OpenTSDB、InfluxDB 和 Facebook 的 Beringei 等系统属于同一类。不同之处在于，Druid 提供了一个支持跨指标聚合的大型分布式缓存层。来看看它与普通的时序数据库有什么不同。举例来说，在 OpenTSDB 中，指标是相邻存储在磁盘上的，如图 5-18 所示。

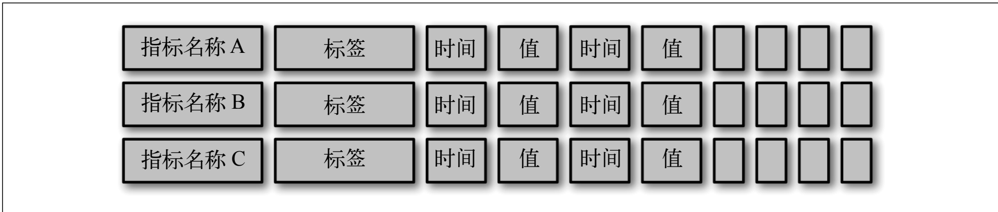


图 5-18：OpenTSDB 的指标存储方式

如果要聚合一组指标，那么就需要先查找它们，然后执行合并连接。在典型的聚合场景中，比如聚合分布式系统的 CPU 使用指标，聚合操作可以被转换成针对数千个甚至数百万个指标进行相加或求平均值操作。在典型的时序系统中，这不仅需要进行大量的磁盘搜索，而且聚合可能不会分布式进行，如图 5-19 所示。

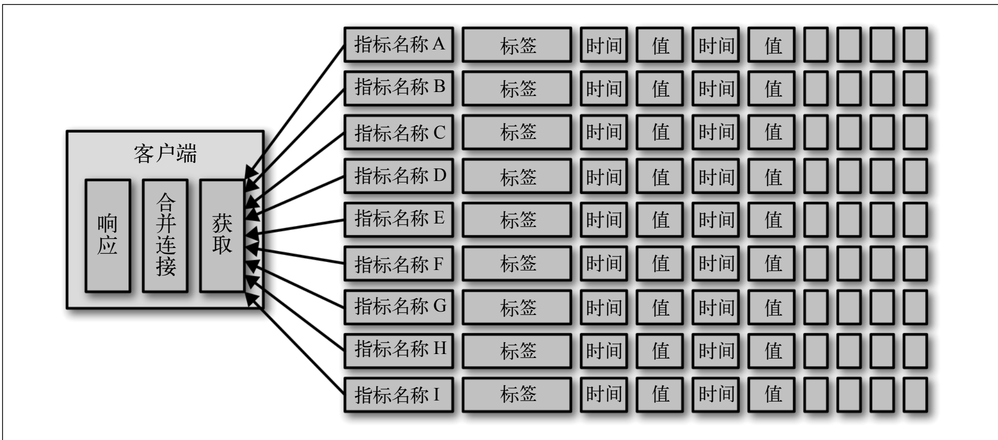


图 5-19：时序数据库中的数据分布

相比之下，Druid 的架构如图 5-20 所示。

需要注意的是，最近使用的数据保存在内存中，支持分布式查询。这避免了磁盘查找，并解决了分布式聚合问题。另外，Druid 利用磁盘存储旧数据，所以它不只是一个内存系统。

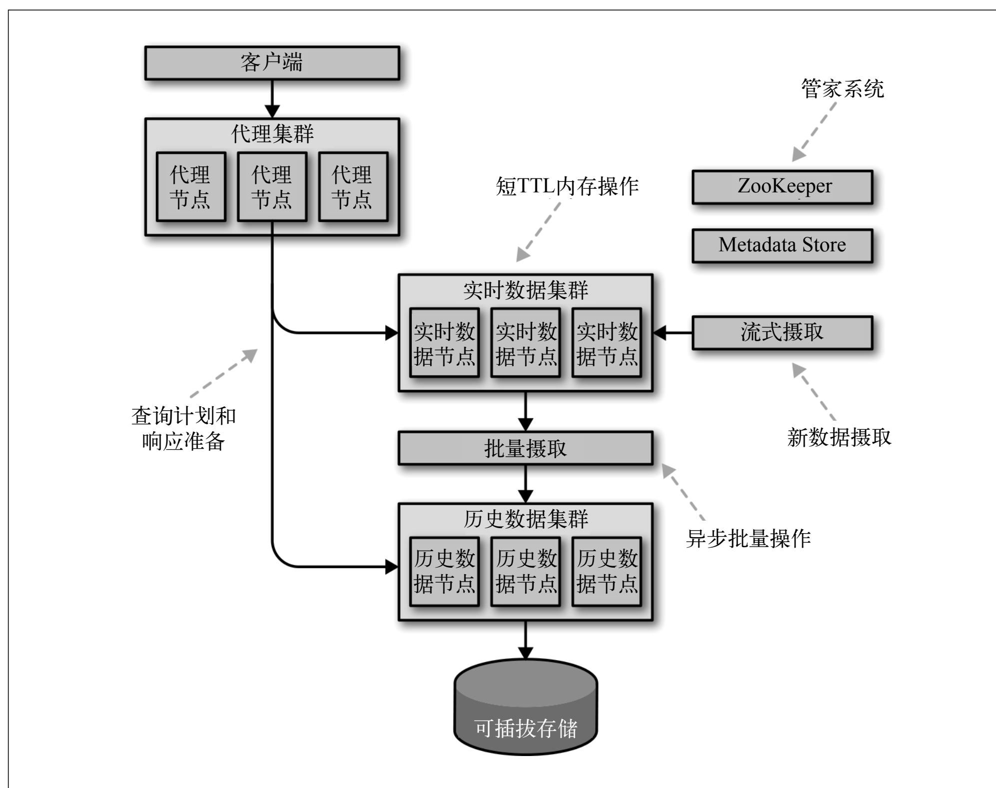


图 5-20：Druid 的架构

2. Redis

Redis 是一个非常流行的内存系统，它提供了多种数据结构。虽然这种内存数据存储系统可能受到节点的成本和数据密度的影响，但也有一些优势，比如支持高级数据结构。

Redis 非常适合作为缓存层，例如，为在线应用程序提供会话缓存。此外，Redis 提供的列表和集合还支持以下功能：

- 添加按分数排名的实体；
- 获取给定实体的排名；
- 从给定排名向上或向下翻页；
- 更新实体并沿着排名顺序移动。

一个常见的示例是动态排行榜，例如，在线游戏得分最高者列表。

在 HDFS、S3 或 NoSQL 中实现这些操作并不容易。尽管 NoSQL 提供了基于索引和数据存储的排序机制，但它在速度方面无法与 Redis 这样的内存数据存储系统相媲美。

5.3 小结

本章介绍了一个用于评估开源企业存储系统的框架。这个框架基于一组标准对存储系统进行分类，并提供了为数据架构选择存储系统的指南。其中的考虑因素包括系统如何对数据进行分区、如何处理数据变更、如何访问数据以及一致性和可用性模型。基于这些考虑因素，我们还定义了这些系统的主要适用场景。

当然，我们无法覆盖所有可用的系统和所有用例，但本章提供的框架应该有助于设计出数据解决方案的最佳架构。

第6章

企业元数据

meta 是一个非常有趣的单词，经常被用在不同场景中。在本书中，它指的是为某些相关信息提供高级上下文和描述的信息。在数据之外的领域，比如电子游戏和电子竞技，经常会听到人们谈论游戏的 meta。在电子游戏中，这个单词是指在游戏中使用的策略和方法。了解 meta 并懂得如何利用它的人比那些纯粹为了娱乐的人能够从游戏中获得更大的成就。

在数据的世界里，元数据（metadata）是与数据以及使用数据有关的信息和策略。元数据提供了额外的可见性，比如存储了哪些数据、这些数据用来做什么以及谁在使用这些数据。在某些情况下，元数据可能比数据本身更重要，而且元数据通常比实际数据存活的时间更长。有效的元数据管理是构建数据解决方案的一个至关重要的部分。因此，本章将讨论如何在数据应用程序中规划和管理元数据解决方案。

在设计数据解决方案时，管理元数据可能是最难以应对的挑战之一。这要归因于复杂的数据系统所涉及的大量数据源、摄取选项、存储系统和访问模式。此外，新的企业数据管理系统能够帮助用户更方便地访问数据，以进行处理和分析，这意味着会不断创建出新的数据集。

不过，一个健壮的元数据解决方案带来的好处通常会超过实现该解决方案本身的难度，而且当下正是规划解决方案的最佳时机。在构建解决方案时就应用策略比在后期再追溯要容易得多。将精力更多地放在项目规划阶段有助于创建出成功且可管理的数据项目。

头发上的花生酱

在 Cloudera 工作期间，我无意中听到了一个有关花生酱和头发的比喻，这些话一直萦绕在我的脑海里。当时我正在与接口设计团队合作，并尝试加快为嵌套类型定义 API 的过程。当时有人说："开发接口就像把花生酱涂在头发上。涂上很容易，但要清理干净就难了。"

可能很难看出这与元数据有什么关系，但可以想象一下从头发上清除花生酱的那种挫败感。在构建一个长期运行的基础设施时，关键的是要做正确的事情。这计划看起来似乎完全与遵循敏捷流程背道而驰。不过，在构建需要持续运行多年的企业数据基础设施时，速度固然重要，但有时候前瞻性也很重要。元数据管理无疑就是一个值得花时间深思熟虑的领域，从而得到一个健壮的流程。

6.1 为什么要关注元数据

在讨论如何管理元数据之前，先来讨论一下为什么要进行元数据管理。成功的元数据策略将在 3 个关键维度上为业务带来价值：数据可见性、数据之间的关系以及与数据相关的监管。

6.1.1 数据可见性

在过去的 10 多年里，我们和很多公司有过合作。我们发现，一家公司是否清楚自己收集了哪些数据以及如何访问这些数据，代表了这家公司数据实践的成熟度。数据目录就是一种元数据，其中包括了字段名、每个字段表示的内容、字段的谱系，等等。公司里所有数据工作者都应该可以访问这些目录（实施了适当的安全限制措施）。

这样做可以获得很多好处。

- 将数据产品更快地推向市场

  只要知道有哪些现成的数据以及它们所在的位置，就不需要浪费时间查找数据或创建新的数据收集管道了。

- 避免重复的工作

  当多个部门或团队需要类似的数据时，重复收集数据会浪费时间和精力。通过访问可用数据的目录可以避免这种浪费。

- 获得更多价值

  要想从数据中获取价值，首先需要弄清楚可以从数据中获得什么价值。随着对收集的数据有了更多了解，数据科学家和分析人员就渐渐知道能够使用这些数据来探索和解决哪些问题。

❑ 找出差距

知道自己拥有哪些数据，与知道自己虽未拥有但应该拥有的数据有着直接的联系。数据的可见性越高，就越容易识别出其中的差距。

6.1.2 数据之间的关系

在提高数据可见性的同时，还可以看到不同数据集之间的相关性。理解这些关系有助于开发出跨不同实体类型的复杂用例。

与硬件和软件位置相关的元数据是系统架构中一种有用的实体关系。为了进一步说明这一点，下面将使用商用 Hadoop 管理解决方案作为示例。我们有一个包含多种实体的数据模型：

- 集群节点；
- 机架；
- 应用程序；
- 服务。

有了这些实体，就可以使用被标记的字段来生成 Hadoop 管理系统的实体关系图了，就像关系模型那样，如图 6-1 所示。

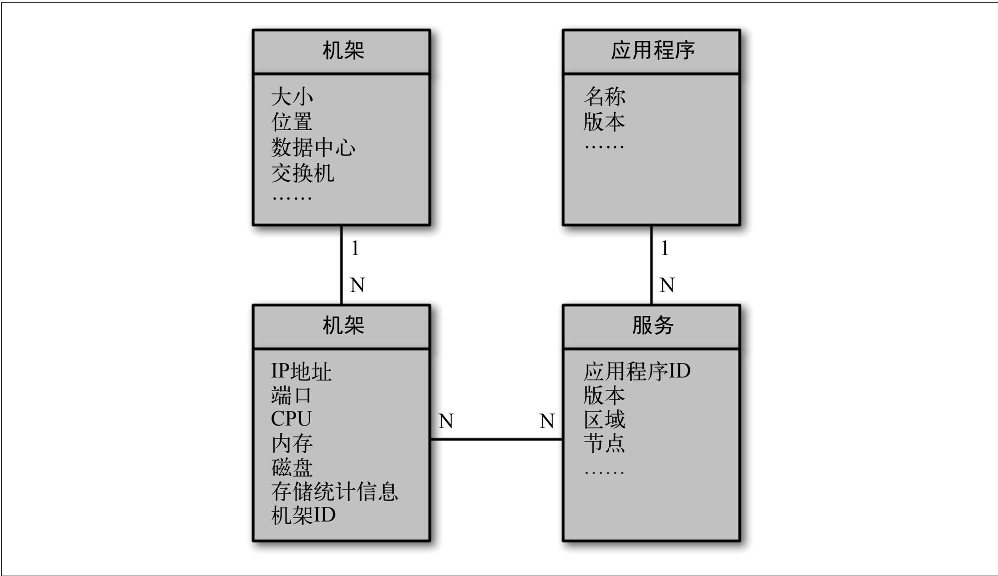


图 6-1：Hadoop 集群的实体关系图

在理解了这些关系之后，可以进行以下事项。

- ❑ 遍历

  沿着任意一条线发现其他实体。可以从某个机架开始，找出在该机架上运行的应用程序。

- ❑ 汇总

  对一组实体的状态或问题进行汇总。比如在上述示例中，可以追踪到关闭一个机架顶部开关会导致多少应用程序中断，因为你知道这些应用程序与受影响机架之间的关系。

接下来，添加更多的实体：

- Kafka 主题；
- 数据库表；
- 查询（包括输入和输出）；
- 流处理操作；
- 源数据生成器。

我们的想法是：只要拥有足够多的实体，就可以了解有关数据的完整故事，比如数据是如何进入系统的、存储数据的位置、如何使用数据、如何利用输出结果，等等。

6.1.3　数据监管

在规划数据项目时，相关的监管问题变得越来越重要。其中有一些促成因素。

- ❑ 数据量

  收集的数据量每年呈指数级增长。这当然也包括了个人数据。

- ❑ 安全性

  随着对个人数据收集的增长，黑客攻击事件越来越多，这影响了数百万人。同时，庞大的数据量赋予了数据更多价值，保护数据变得越来越困难，非法获取数据和滥用数据的情况正在增多。

- ❑ 权力

  数据能够给公司和政府带来的好处是显而易见的。现在的问题是：在数据收集和使用方面，公司和政府应该获得多大的权力？随着此类问题变得越来越紧迫，对数据进行监管和监督的呼声将会越来越高。

监管回应

有些监管机构已经开始对数据收集的负面影响做出回应。例如，欧盟通过的《通用数据保护条例》（GDPR）要求公司必须制定适当的政策来应对以下问题。

掌握个人信息的权利

任何人都可以要求任意一家公司提供有关自身数据的详细信息。这对数据系统的影响可能

会非常大。不仅需要知道某个人的数据保存在公司的什么地方，还需要弄清楚如何以有效的方式查询数据，以及如何在不泄露内部数据系统的机密信息的情况下显示数据。

删除个人信息的权利

任何人都可以要求删除与其相关的数据或对数据做出修改。这也可能对系统产生重大影响。

限制数据应用的权利

公司只能将收集到的数据用于某些特定的目的，并需要在个人同意公司提供的服务条款之后才这样做。这也将影响到数据系统，因为它要求公司收集那些使用了特定数据的处理类型的信息。

安全性影响评估

企业有必要快速知道哪些数据在遭遇黑客攻击时受到了损坏。了解收集了哪些数据及其存储位置对于这类评估来说至关重要。这涉及前面提到的所有考虑事项：有关数据的知识、数据之间的关系、理解与数据有关的访问和处理情况，等等。

拥有元数据对于满足上述监管要求至关重要。如果不知道从某个客户那里收集了哪些数据以及存储于何处，那么就无法遵守“删除个人信息的权利”规则。

6.2 数据架构中的元数据类型

前文已经讨论了元数据的重要性，并说明了为什么需要一个好的元数据策略。接下来讨论在制定可行的策略时需要关注哪些元数据类型。

- 静态数据

  静态数据是已经被摄取到磁盘存储（在某些特殊情况下是内存）上的数据的元数据。可以是长期存储的，例如 Hadoop 分布式文件系统（HDFS）；也可以是短期存储的，例如 Kafka 主题。

- 动态数据

  一般来说，动态数据是通过数据管道传输的数据。

- 数据源（实体）的元数据

  数据源（实体）的元数据包括在 6.1.2 节中讨论的实体类型。

- 有关数据处理的元数据

  有关数据处理的元数据，也可以将其称为操作型元数据。

- 报告和仪表盘

  报告和仪表盘是描述系统生成的报告的元数据。

6.2.1 静态数据

静态数据是指长期保存在系统中的数据，例如数据库中的表和字段、基于 Lucene 的系统中的集合、时序数据库中的指标，以及文件系统或对象存储中的文件。可以将这一层的元数据视为可用数据的目录。

在存储这类数据的元数据时，通常会使用有意义、可搜索和可索引的数据对其进行标记。来看一下客户收据数据表示例（表 6-1）。数据库的名字叫作 Purchasing。

表6-1：客户收据数据表

字 段	类 型
User_id	Long
Receipt_num	Long
Item_purchased_id	Long
Amount	Decimal(7,2)
Timestamp	Timestamp
Method	String
Card_id	Long
Purchased_port	String

表 6-1 中显示的每个字段和类型都是数据库、表和字段的元数据。至少，我们需要保存这些元数据，并能够访问它们。可以将它们保存在基于 Lucene 的系统或像 Hive Metastore 这样的元数据存储库中。也可以捕获更多的元数据，以下举例。

- **审计日志**

  数据表是如何创建的，以及表的变更历史记录。

- **注释**

  对数据表和字段的注释。

- **标记**
  - 标记字段，假设它们包含个人数据（如 `User_id` 和 `Card_id`）。
  - 标记可连接表。假设 `Card_id` 是唯一标识符，因为要在查询中隐藏实际的卡号。为了确保数据库用户知道如何通过执行连接来获取卡号，需要标记 `Card_id` 字段，指明需要连接哪个表来获取真实的标识符。
  - 详细信息，如数据所有者。

- **访问权限**

  谁可以访问数据表。

- **使用情况**

  哪些查询和用户正在访问数据表。

- ❑ 来源

  数据的谱系，例如，数据是从源系统收集的还是由转换作业生成的。

6.2.2 动态数据

我们已经讨论了静态数据的元数据，接下来介绍动态数据的元数据。在现代数据架构中，可以通过多种方式摄取数据，来看看其中的一些方法。

1. 批次传输

在批次传输模式下会定时发送数据，然后将其以记录批次的形式存入系统中。批次传输的常用方法是通过 FTP、SCP 或切换到对象存储。数据可以采用 CSV、JSON、二进制等格式进行传输，而且通常会被压缩，如图 6-2 所示。

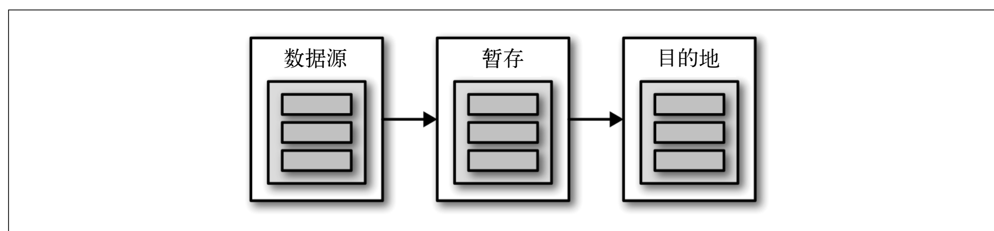


图 6-2：批次数据传输

2. 流式传输或微批次传输

在采用流式传输或微批次传输时，数据会在系统中持续移动。物联网中的传感器数据就是一个例子。通常，管道中的某个位置会存在某种形式的可扩展管道（例如消息队列或 Kafka），后面会有一个服务，对流式数据进行转换并将其摄取到目标位置，如图 6-3 所示。

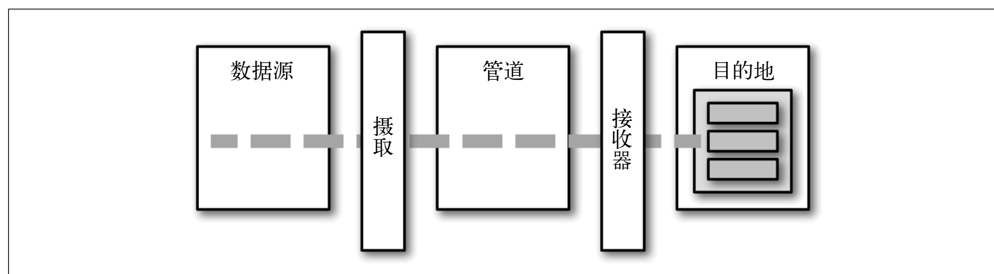


图 6-3：流式数据传输

3. 应用程序操作

在这个架构中，数据存储直接连接到应用程序，应用程序直接将数据写入这些数据存储。不过，操作数据存储与分析数据存储相分离的情况变得越来越普遍。这意味着即使应用程

序有一个操作数据存储，它也会使用批处理或流处理将数据推送到外部的分析数据存储。这种工作流如图 6-4 所示。

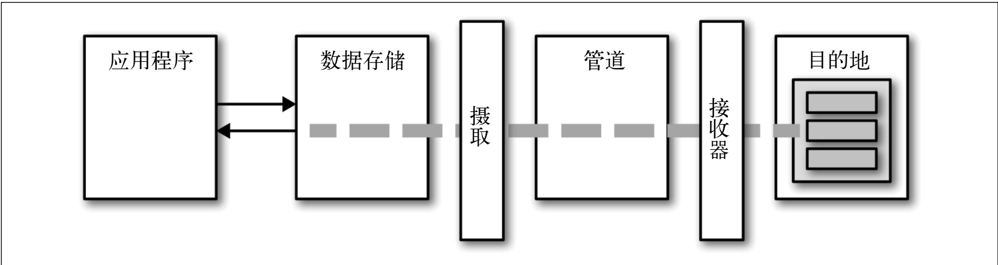


图 6-4：应用程序和数据存储之间的数据传输

4. 转换后处理

转换后处理发生在基于现有数据集创建新数据集的时候（图 6-5），通常会使用 SQL 或 Apache Spark 等分布式处理引擎来进行转换后处理。这些处理作业既可以是生产作业，也可以是用户在临时会话期间触发的作业。

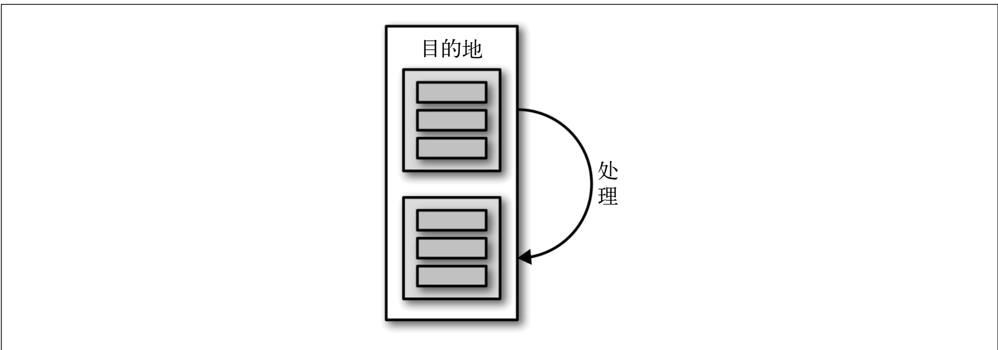


图 6-5：基于现有数据集创建新数据集

5. 为动态数据捕获元数据

数据管道有很多入口点。我们需要了解所有入口点，并对它们进行分类，以成功地捕获元数据。了解存储在系统中的数据的谱系十分重要。接下来深入探讨一下在收集有关数据移动和数据转换的元数据时需要捕捉的内容。

路径

我们要标记数据流经系统的路径。图 6-6 显示了系统路径包括很多组件，例如源系统、数据收集系统、数据路由、数据转换，等等。后文将讨论与某些组件的特定元数据相关的注意事项。

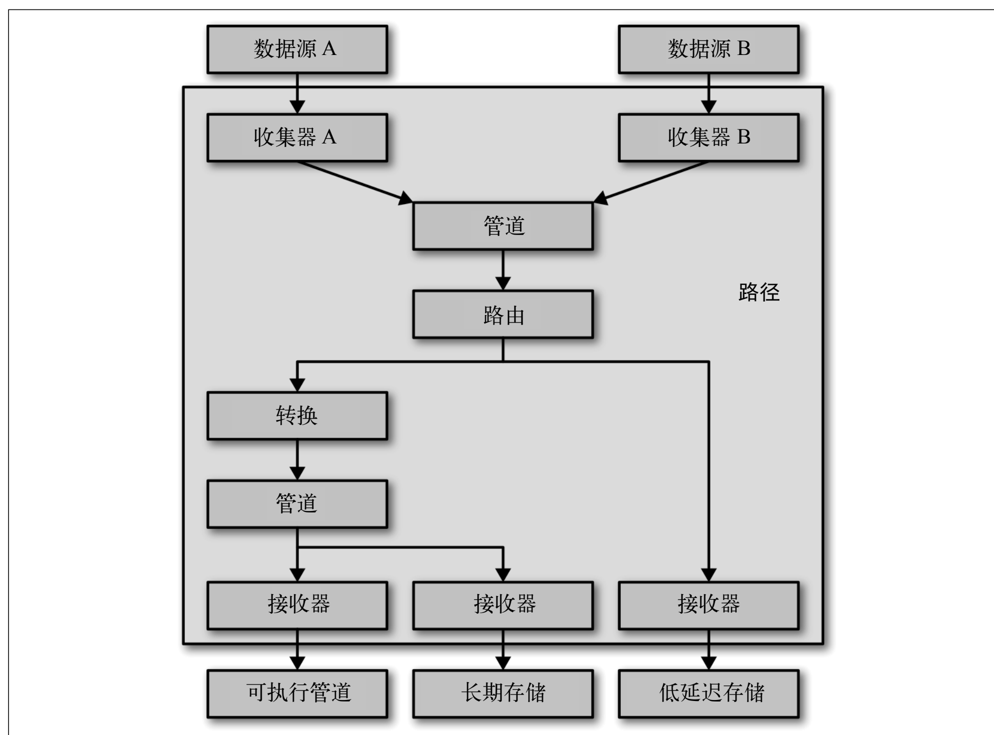


图 6-6：数据管道路径示例

数据源

数据源是指数据的来源。它们通常是外部系统，也可以是内部系统或现有的数据源。

数据移动

数据移动定义了数据采取的路由，通常会描述数据是分叉的还是被发送到多个位置的。例如，图 6-6 展示了初始管道会发送给数据转换组件以及接收器和目标存储系统。

第 7 章将讨论更多有关路由的内容，现在可以先将路由看成是数据从数据源到目的地的路径。图 6-7 展示了一个基本示例，通过 Kafka 从多个数据源接收数据，然后根据某些标准（如数据的类型或数据的预期用途）将输入数据路由到多个目标存储系统。

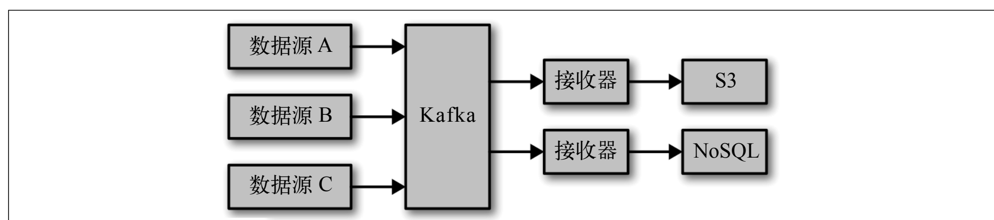


图 6-7：路由多个数据源

在图 6-7 中，路由元数据告诉我们来自数据源 B 的数据将通过 Kafka 到达 S3 和一个 NoSQL 系统。这只是一个简单的示例，在现实世界中，路由可能还会包括数据过滤、数据修改等。

转换

这是关于如何转换数据的文档。以下是需要记录的一些因素。

- 格式变更

  源数据的底层数据结构是什么样的，在转换之后又是什么样的？例如，源数据可能是 JSON，但目标格式可能是 Protocol Buffers。

- 数据保真度

  数据是否经过过滤或修改，使其不再具有完全的保真度？

- 原子、连接、会话或上下文

  转换是否只基于输入记录，或存在来自外部数据的其他上下文？

- 作业元数据

  有关处理作业的基本信息：名称、使用的技术、生产状态、版本号、项目负责人、SLA，等等。

- 字段元数据谱系

  本章在前面部分讨论了字段元数据和标记。在将字段输入到转换中以生成新字段时，需要追踪标记和元数据的谱系。包含个人身份信息（PII）的字段就是一个很好的例子。如果字段是由转换生成的，并且数据源当中包含了 PII 字段，那么就应该将历史 PII 数据的存在捕获到新字段中。

目的地

目的地记录了数据的着陆区。这应该与前面讨论过的静态数据的元数据保持一致。实际上，应该将这些元信息添加或链接到目标元数据。

6.2.3　数据源的元数据

正如目的地需要元数据的存在一样，数据源也需要有元数据。如果现在你只有一个数据源，这似乎有点夸张，但通常情况下你会收集多个来源的数据：手机、服务器、传感器、数据库，等等。我们希望像捕获系统中的元数据一样捕获这些数据源的元数据。

以下是捕获这些元数据的一些原因。

- 了解是什么在向系统提交数据

  在手机世界里，手机的类型（型号、内存、CPU 等）或位置（信号强度、与移动通信基站的距离）等信息可能有助于了解如何改进服务。

- 识别恶意活动

  当你拥有数百万个数据源时（例如在物联网场景中），可能会有人想攻击你的系统。了解数据源有助于想出解决方案来识别这些不良分子，并将它们与真实的数据源分开处理。

- 了解关系

  假设你正在监控硬件和查看故障影响。在这个时候，需要知道服务器与机架和交换机之间的关系元数据以及位置元数据。这也适用于物联网场景，比如在汽车、飞机等大型系统中，从多个传感器获取输入。

6.2.4 有关数据处理的元数据

系统中的数据处理操作各式各样，从简单的 SQL 查询到高级的 Apache Spark 作业、机器学习，等等。此外，由于处理技术的多样性，很难知道哪些元数据正在使用以及用于什么目的。至少需要确保得到了基本的信息，从而为构建系统奠定基础。

- 访问

  谁在访问特定的数据？

- 频率

  访问数据的频率是怎样的？

- 输出

  数据是发送到系统之外还是发送到系统中的另一个数据表中？

- 使用的技术

  一些技术会留下清晰的审计线索，例如 SQL，但其他技术可能很难追踪，例如 Apache Spark 作业。Spark 作业的逻辑隐藏在 Java/Scala/Python 代码中，很难用审计工具来解释。

请注意，捕获这些元数据对于生成实际、可操作输出结果的生产作业来说更为重要。一方面，对于那些用于测试、研究等场景的作业，需要确定元数据的合理捕获量。另一方面，GDPR 等法规不区分生产环境和开发环境，因此在规划时需要牢记这一点。

对于生产作业，可以考虑提供源代码的访问权限。例如，把代码放在公司的 GitHub 仓库中，这样就可以对代码进行评审。

随着算法变得越来越复杂，理解这些算法如何做出决策也变得越来越困难，因此，有必要特别考虑一下机器学习作业。越来越难以理解的模型，导致了更多不良甚至不道德的结果出现，从而让公司面临风险。请考虑捕获以下内容。

- 模型的目的

  模型的目标用例是什么？

- ❑ 使用的技术和算法
  例如，用于执行 k 均值聚类作业的 Spark MLlib 或 TensorFlow。
- ❑ 输入特征
  可以将特征看作机器学习算法的输入。例如，从输入数据集中选择特定的数据属性，作为 k 均值聚类的输入特征。
- ❑ 训练数据集和测试数据集
  用于训练、验证和测试模型的数据。
- ❑ 训练目标
  用来评估模型是否成功。这些目标将高度依赖于应用程序的需求、算法等，例如分类模型的分类准确性。
- ❑ 人为调整模型
  某些模型可以通过手动输入进行修改。对于这种模型，最重要的是捕获调整了哪些内容以及背后的原因。
- ❑ 模型负责人
  负责实现、定义已部署模型的人是谁？

6.2.5　报告和仪表盘

常见的输出是人类可读的报告和仪表盘。创建它们很容易，但很快就会过时。

作为元数据策略的一部分，为了获取可见和可操作的报告，至少应该捕获以下内容：

- 数据源；
- 数据转换；
- 有关报告创建者的信息；
- 日志修改；
- 报告的目的；
- 表示它与哪些内容相关的标签。

标签有助于将报告映射回报告所涉及的内容，例如与区域、位置、版本等相关的标签。因为一个组织可能会有数百甚至数千个仪表盘，所以标签可以作为一种将仪表盘链接回它们所报告的内容的工具，从而减少重复报告的生成。

6.3　元数据收集

元数据收集是一项极具挑战性的任务。事实上，很多公司不具备捕获元数据的有效流程。

从元数据管理来看，公司通常可以分为以下类别。

- 多数据孤岛

  这类公司有很多部门，每个部门在元数据管理方面具有不同的成熟度，通常很少有或者根本没有跨部门可见的元数据。

- 集中的知识

  在这类公司中，只有少数人可以访问或了解数据集。

- 周期性普查

  这类公司的数据目录是手动维护的，而且几乎总是过时。每年或者每隔几年，他们会派人对组织中存在的数据集进行编目和记录，并生成数据库甚至电子表格，但这些很快就会过时。

- 深思熟虑

  这类公司从一开始就制定了收集数据和生成数据的策略。由于公司的政策和流程，所有元数据都尽可能保持最新。本节将以这类公司作为范例展开讨论。

接下来深入研究两种用来制定元数据收集策略的常用方法：声明和发现。**声明**是指通过使用系统的正常操作来收集数据，**发现**则是为了弄清楚事实背后发生了些什么。

6.3.1　声明式元数据收集

声明式元数据收集是指在向系统添加新数据源、更改数据用途或移动数据时创建元数据。这样就可以从参与数据收集的个人或组织那里收集元数据。

只有当用于装载数据、管理数据路由或用途的工具在报告变更时，才会发生声明式元数据收集。这很可能是通过将这些操作限制在经过批准、完全集成到元数据解决方案中的路径实现的。实现这一目标的方法依赖于特定的工具。

声明式元数据收集需要将数据操作限制为已批准的路径。因为没有人喜欢被迫严格遵守流程，所以这看起来有点累赘。可以通过启用一些流程来促进这些元数据的收集，以避免使其成为一种负担。此时，有必要回顾一下那些更具挑战性的数据管道特性。

- 它们并不简单

  要正确地执行摄取数据和处理数据等操作，需要对数据进行监控、转换、扩展和故障转移等。

- 它们可能会使用相对标准化的架构

  我们讨论了批次和流式的用例，虽然它们的实现方式不同，但完全可以使用有限数量的实现来支持大多数用例。

简化数据管道的使用有助于将声明式元数据收集集成到管道中。可以通过以下方法来简化数据管道的使用。

- ❑ 通过供应商工具来声明摄取

  例如，可以使用 Confluent Schema Registry 捕获所有与 schema 相关的信息，并为元数据解决方案提供信息（作为 schema 注册的一部分）。

- ❑ 使用 Hive Metastore

  如果将声明式方法与元数据解决方案集成在一起，并将报告变更作为变更过程的一部分，那么使用 Hive Metastore 或其他集中式 Metastore 可能是声明式方法的关键。

声明式解决方案的主要思想是在完成任意操作时触发元数据系统，这些操作可能是摄取数据集、基于数据集创建新数据表或者转换数据集。在触发元数据系统后，供应商工具会将通知作为系统操作的一部分发送给元数据系统，或者元数据系统会读取管理系统的编辑日志。然而，这种方式的实现将取决于具体的解决方案。

应该声明什么样的元数据

在设计声明式路径时，需要构建之前讨论过的元数据类型：

- 数据源；
- 使用的技术；
- 表、字段名和标签；
- 目的地；
- 数据的使用；
- 数据的所有者或管理员。

最接近数据的人就是那些输入数据的人。他们最了解数据，而且可能与实现完整而准确的元数据收集有关。这也是一个获取其他链接数据集的知识和上下文以及避免潜在的数据重复的好机会。

6.3.2 发现式元数据收集

在某些情况下，数据不是通过声明的方式创建的。这可能是由于收集过程或工具无法提供全面的解决方案。因此，与声明式解决方案不同，发现式解决方案关注的是最终的数据，并尝试使用脚本模式找出发生了什么事情。

声明式过程很难覆盖到的一个情况是：通过 SQL 或其他 ETL 工具，基于其他表创建新的表或基于其他数据源创建新的数据集。在这种情况下，发现式元数据收集是一种更好的办法。通过来自日志或作业的信息找出输入和输出，还可以扫描现有的表，看看它们如何与已知表重叠。

如果发现在定义方面存在差距，有以下 3 种选择。

- ❑ 评审数据表的元数据

  表中的标记或注释可以体现更多有关表及其来源的信息，可能还会体现所有权信息及其与数据库的关系。有了所有权信息，我们就知道在发生信息丢失时应该联系谁或通知谁。

- ❑ 字段内容分类

  目前有很多工具使用正则表达式和机器学习对表中的数据进行采样，并且可以体现表中的信息。例如，标注社保号、信用卡号和电话号码。这些工具不仅可以用来添加列或表的元数据，还可以告诉我们表的谱系信息。请注意，其中大多数工具可能来自供应商和第三方工具。

- ❑ 审计追踪发现

  与数据字段分类类似，有些工具可以通过日志、生成表等元素来推断元数据。这样可以在没有显式声明的情况下为数据、谱系发现以及文档（如表和数据集）创建审计日志。

如何处理未记录的数据集

我们讨论了几种方法，它们可用于发现未记录的数据集的源头。但是，在某些情况下，可能无法完全发现源头，或者会发现一些不可接受的结果。对于这些情况，也有一些办法。

- ❑ 联系数据所有者

  如果有足够的信息找到数据所有者，可以通过电子邮件、消息或警报与他们取得联系，从而收集更多的信息或做出修改。

- ❑ 锁定数据表

  在某些情况下，处在未记录或已记录状态下的数据集是不可接受的。这个时候可以锁定数据表，除了管理员团队，没有人可以访问它，甚至数据集所有者也无法访问。

- ❑ 删除数据

  在锁定表后，最好可以启动存活时间（TTL）计时器，TTL 到期后就删除数据。

- ❑ 审计追踪

  需要记录和检查所有的通知、锁定和删除操作。还应该针对违反规则的用户采取纠正措施。

6.4　元数据管理实践

前文已经详细讨论了元数据的重要性，以及在为项目定义元数据策略时应该规划些什么。如何实现全面的元数据策略则是一个更大的挑战。这一挑战很大程度上来自大量的系统、数据源、数据格式等，它们很可能是分布式数据管理系统的一部分。可惜的是，要找到一个可以跨整个数据架构管理元数据的工具，仍然是一个挑战。

应对这一挑战的一种方法是创建自己的解决方案。当然，在实现和维护自定义解决方案时会带来另外的挑战。更好的办法是尝试使用供应商提供的或第三方的解决方案。大多数企业数据管理供应商的解决方案都会提供元数据定义、数据谱系追踪、数据审计等功能。这些供应商在产品中提供了适当的解决方案，可以作为解决元数据管理问题的有力工具。寻找跨产品的解决方案是一项更大的挑战，不过，一些数据集成供应商承诺提供“单一管理平台”来管理跨系统的元数据。需要通过探索数据架构所依赖的供应商或项目的解决方案（包括第三方解决方案）来制定有效的元数据管理方法。

6.5　小结

在本章中，我们讨论了将元数据收集作为数据架构核心部分进行计划和实现的重要性。原因有很多，包括监管方面的原因和系统可维护性方面的原因。成功的元数据策略有助于确保系统的成功。一个成功的元数据策略应该包含以下内容。

- 确定需要进行元数据管理的数据集，包括静态数据（长期存储或短期存储）、处理管道中的动态数据、源数据以及数据处理和分析。
- 定义不同数据集需要捕获的元数据。
- 捕获元数据的方法，包括声明式元数据收集和发现式元数据收集。
- 识别用于管理元数据收集的工具——供应商提供的或第三方的解决方案。

未能实现可靠的元数据策略可能会导致以下结果。

- 用户无法找到所需的数据。
- 数据进出系统的非标准机制。
- 有价值的数据闲置，因为甚至没有人知道它的存在。
- 潜在的法律或监管行为可能对公司造成损害。

我们还讨论了某些元数据类型的特殊情况，包括机器学习模型和敏感数据，如个人身份和财务数据。最后，我们讨论了如何实现这些目标，包括使用支持声明式元数据收集或发现式元数据收集的工具，以及如何找到可以辅助元数据收集的工具，包括供应商提供的和第三方的工具。

本章的主要结论是，我们应该像对待公司中其他数据集一样对待元数据。通过关联、连接和分析从元数据中获取价值，毕竟元数据代表了公司的运作方式和对数据的处理方式。数据是当今企业最重要的资产之一，因此捕获有关数据收集、数据存储和数据使用的信息也同样重要。

第 7 章

确保数据完整性

在使用开源企业数据管理系统时，在数据架构中使用多个存储和处理层是很常见的。为了优化访问，通常会使用多种格式存储数据。这意味着可能会存在重复数据。在过去，由于费用和复杂性方面的问题，这可能会被视为一种反模式，但随着新系统和廉价存储的出现，这种方式反而变得切实可行。

当数据从数据源移动到最终存储时，需要确保数据完整性，这一点始终未变。**数据完整性**是指数据在整个数据管道中的准确性和一致性。为了确保数据完整性，必须知道所有流经系统的数据的谱系。

本章将讨论数据完整性，并提供一些示例来说明在数据流经系统时如何确保数据完整性。我们将讨论全保真数据，即保留了源数据完整上下文的数据。这些数据可能与源数据的存储格式不同，但只要能够返回到原始状态，它就被认为是完全保真的。我们还将讨论从原始源数据派生出来的数据集，例如，经过过滤和聚合的数据。无论最终的数据集是完全保真的还是派生的，保持数据完整性都至关重要。

数据在系统中移动时所发生的处理类型可以用来确定数据是完全保真的还是派生的。为了更清楚地说明这一点，下面提供一些全保真数据集和派生数据集的例子。

- 全保真数据集
  - 使用无损压缩格式压缩的数据——这些是完全保真的数据，只是被压缩了。
  - 数据从一种格式转换为另一种格式，例如，从 JSON 到 protobuf——这些也是完全保真的，只不过格式不同。

- 派生数据集
  - 被过滤了列的数据——这是派生（子集）数据，因为保留了原始数据，但过滤了特定的值。
  - 被过滤了行的数据。
  - 聚合数据。

并非所有数据集都刚好属于某一个类别。例如有一个数据集，它将原始数据集与附加的聚合列组合在一起，在这种情况下，原始列的数据具有完全保真度，但新列是派生数据。

本章不是要试图涵盖每种数据处理场景，而是要提供一些示例，帮助你思考如何设计自己的数据管道，以确保系统数据的完整性。首先介绍一些可能会被用在系统中的数据管道类型，并描述如何从数据完整性的角度来设计。

7.1 构建数据管道

在讨论具体的示例之前，有必要将数据管道设计分为两类：预定义路径和发现路径。

预定义路径是指在实现数据管道之前的分析和设计阶段得到的数据管道。通常，预定义路径更容易监控和审计，因为数据的处理和建模是在管道实现开始之前定义的。这样就可以规划数据在流经管道时需要进行哪些监控。

与之相对的是发现路径，它可以被看作一种临时或自助的数据处理方法。任何人都可以在原始数据或选定的派生数据集上构建任意的路径，然后利用软件来追踪谱系，这样就可以在事后通过审计来确定数据的有效性。这种方法的问题在于，它很快就会变得混乱不堪。如果业务中包含了大量的审计，建议谨慎使用这种方法。

作为一个真实示例，可以考虑使用 Apache Hive 在 Hadoop 存储的数据之上创建数据仓库。有了 Hive，数据工程师或管理员就可以基于源数据创建数据库，然后将这些数据库提供给用户做数据分析。接着，用户基于这些数据库创建自己的数据集或派生表。这样做的优势是用户可以灵活地使用数据，但也会导致数据源和价值未知的表和数据集的数量激增。

图 7-1 提供了预定义选项和发现选项的直观描述，并显示了它们在生产力与可管理性图谱中所处的位置。

我们很可能对处于中间位置的路径更感兴趣。这需要一些能够灵活变化的预定义路径，并创建沙盒，让其中的数据可以自由移动。实现这一目的的诀窍是使用验证和接口点。第 4 章讨论了接口，稍后将讨论更多有关验证的内容。现在，先来深入研究预定义路径和发现路径。

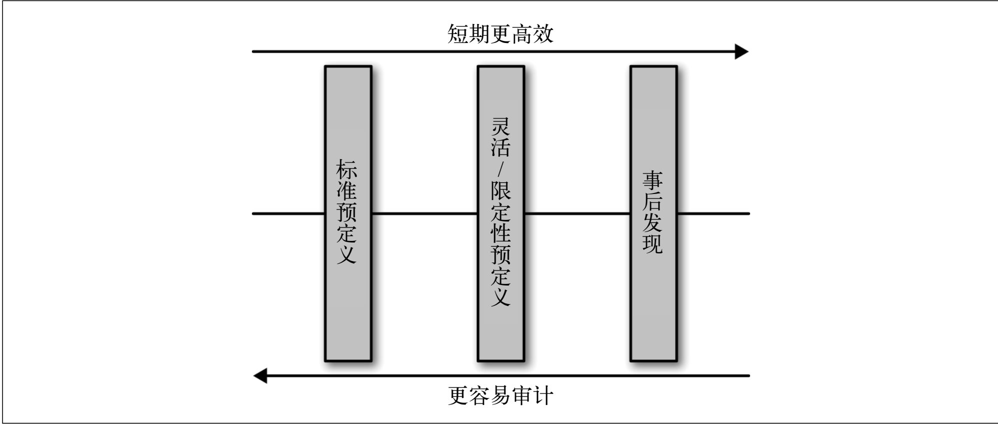


图 7-1：预定义路径与发现路径

预定义数据管道

预定义管道需要提供访问点，通过这些访问点将数据移动到系统中，然后再使用正确的格式将它们推送给正确的目的地。预定义管道在现实世界的示例，可以考虑一下家中的断路器和电路，如图 7-2 所示。

请注意，断路器并不关心电源的细节，它只关心电源是否符合特定的标准。然后，断路器和电线将电输送到插座。最后，插座上的设备按照预期的标准消耗电力。

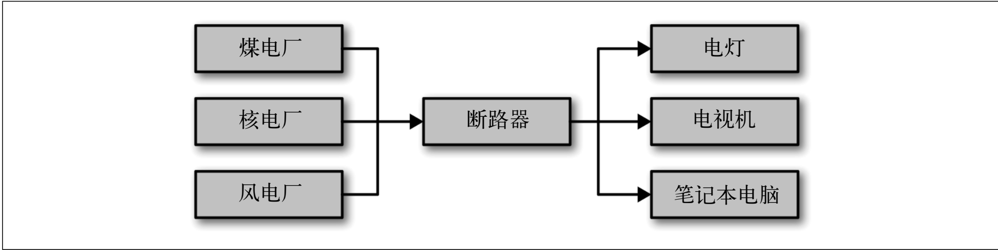


图 7-2：家庭电路示例

同样，预定义管道需要具有以下属性：

- 定义良好的输入接口和输出接口；
- 出问题时的故障转移机制；
- 保证级别。

接下来看两个预定义管道的实际实现，它们能够确保数据完整性：一个用于处理批次数据，另一个用于处理流式数据。

1. 批次管道

典型的预定义批次管道如图 7-3 所示。请注意，这个示例假设了一个基于 Apache Hadoop 的环境。

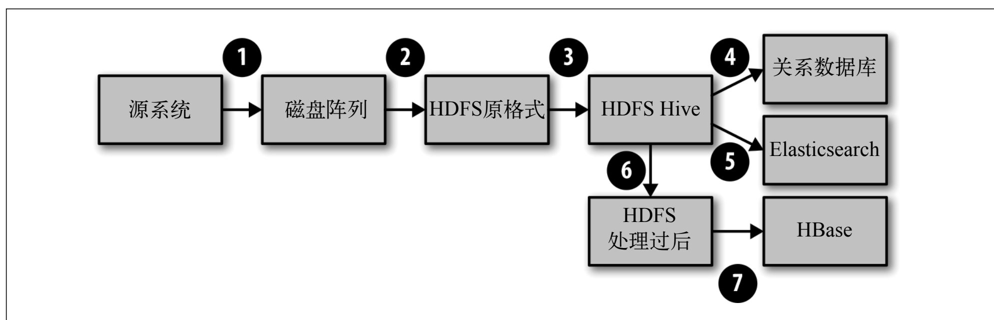


图 7-3：批次管道

下面一步步解释。

(1) 来自源系统的数据被移入数据管道。在这一步，数据只是被传输到临时存储数组。在某些情况下，数据可以直接进入长期存储系统（在本例中为 HDFS），但需要注意的是，它们仍然是存储在单一系统中的原始数据。

(2) 如果数据还未进入长期存储（在本例中为 HDFS），那么就在这一步将其移到长期存储中。此时，文件仍然与源文件 100% 相同，唯一的变化是这里数据被当作数据块在 HDFS 节点之间复制。

(3) 在这个阶段，使用 MapReduce 或 Spark 等处理框架为数据添加结构，并将数据保存成 Apache Hive 的优化格式（如 Parquet 或 ORC）。这可能只是格式更改，但根据输入数据的不同，也有可能会进行过滤。如果没有进行过滤，那么它们仍然是完全保真的数据。但是，如果进行了过滤，那么结果数据将成为源数据的派生数据子集。

(4) 数据被移动到关系数据存储中。请注意，在之前的步骤中已经为数据添加了结构，因此除了格式更改和可能的过滤之外，不应该有其他更改。

(5) 将数据从 HDFS 批量加载到 Elasticsearch 中。同样，这里不应该有数据更改，只有格式更改和可能的过滤。

(6) 针对结构化数据运行会话作业。这个步骤的结果数据不再是完全保真的，而是源数据的派生数据集。

(7) 将会话数据从 HDFS 批量加载到 HBase 中。这一步不应该出现任何数据更改，只有格式更改和可能进行的过滤。但是，由于它所读取的数据不再是完全保真的，因此 HBase 中的数据也是派生数据。

在图 7-3 的基础上加入数据在管道中的移动状态，如图 7-4 所示。

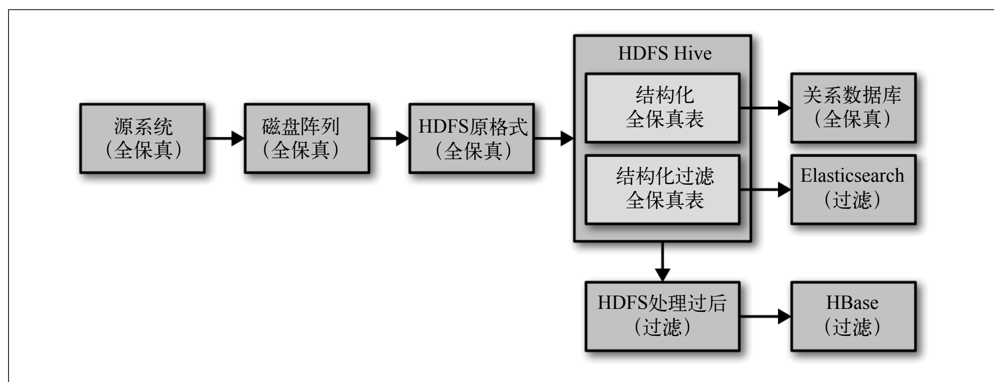


图 7-4：批次管道的细节

2. 流式管道

预定义流式管道可能如图 7-5 所示。请注意，这个示例增加了 Apache Kafka。

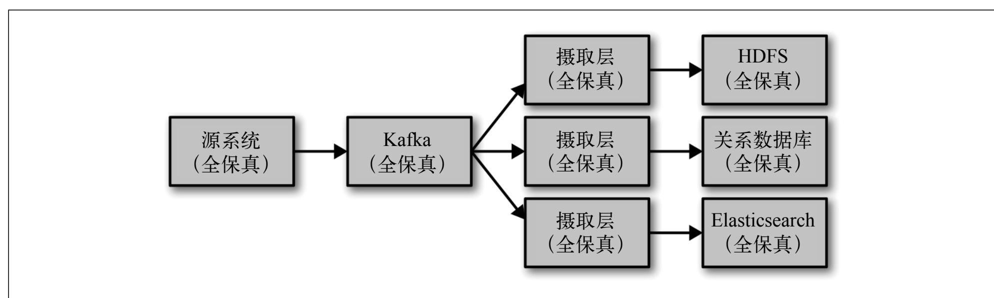


图 7-5：流式管道

这里有 3 个独立的存储系统，每个存储系统以相同的顺序获取相同的数据（对应 Kafka 分区）。这意味着数据在整个摄取路径中仍然保持了完全保真度。

与批次架构一样，我们希望在数据进入存储之前对数据进行增强和过滤。对于这种情况，有几种选项。第一个选项是在摄取层进行增强和过滤，如图 7-6 所示。

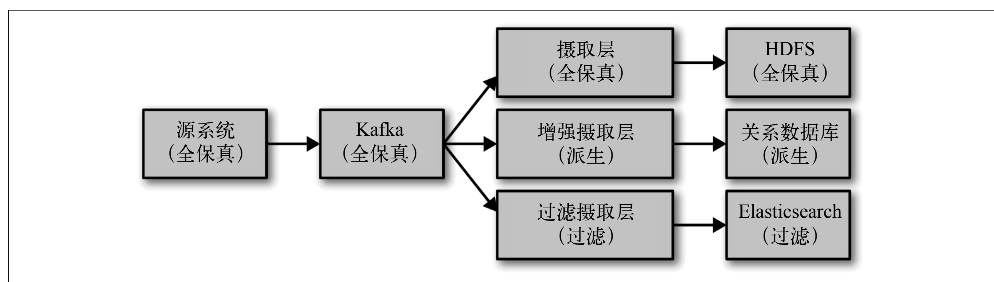


图 7-6：具有过滤和增强功能的流式管道

也可以添加额外的 Kafka 层，如图 7-7 所示。这样做的优点如下。

- 分区

  以特定的方式对数据进行重新分区，从而优化存储和处理。

- 路由

  具有高级的路由逻辑，如果将它们包含在摄取层中，效率会很低。

- 变更

  能够对多个数据集进行数据增强，这些操作最好在摄取之前进行。

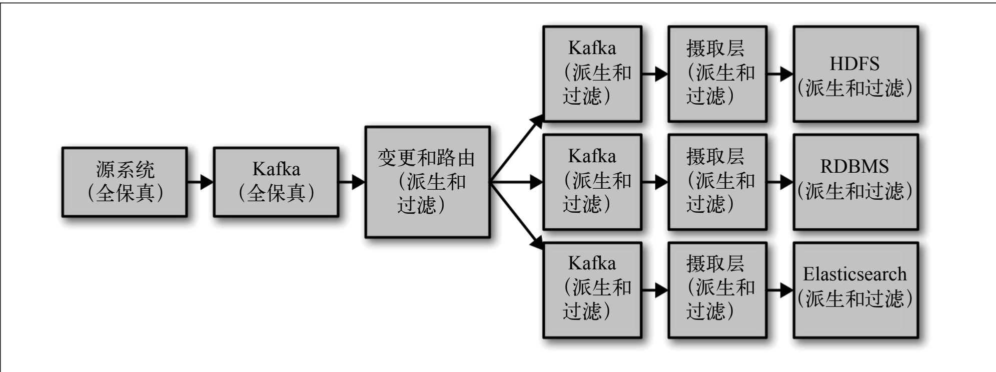


图 7-7：包含 Kafka 的流式管道

需要避免为多个目的地提供单个摄取服务，如图 7-8 所示。

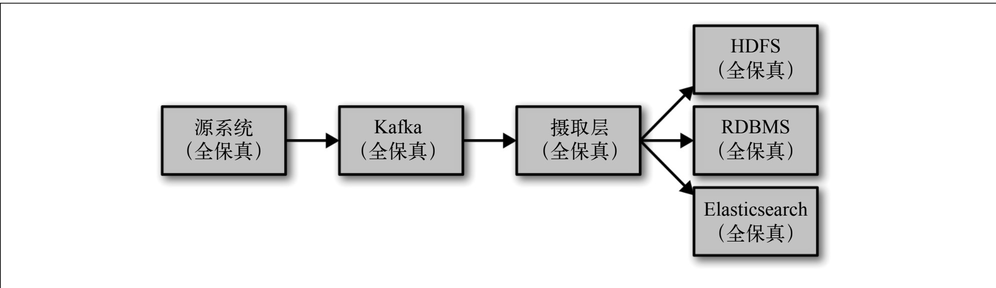


图 7-8：具有多个目的地的流式管道

这样做的缺点如下。

- 单体

  这些存储系统可能有自己的依赖项和库。因此，通过一个进程向所有目的地写入数据将增加出现库冲突的概率。

- 复杂性

  实现越是简单、越是集中，测试和维护就越容易。记住，越简单越好。

- 性能依赖

  摄取性能会受到最慢的存储系统的约束。

- 系统性故障风险

  存储系统故障意味着所有数据都将停止流入其中的任何一个系统。

- 不同的批次大小

  不同的存储系统按照不同的写入方式进行优化。HDFS 和 Amazon S3 需要较大的批次，而其他类型的系统在采用较小批次时表现更好。

- 难以回滚

  假设存储系统出现问题，需要从一个检查点恢复。如果这个时候向所有系统写入数据，那么恢复过程将会变得非常复杂。

- 版本升级

  升级到一个存储系统可能需要重新启动所有摄取节点。

7.2 验证数据管道

如果仔细阅读，你会注意到，7.1 节描述的所有路径都涉及某种转换。

- 摄取

  将原始格式转换为目标系统使用的格式。

- 增强

  以某种方式转换或添加数据。

但是，在这些步骤中发生的转换可能会引入一些错误，破坏数据完整性。那么，应该如何通过测试来确认可以信任系统呢？来看看 4 个选项：行数、唯一计数、全字节比较以及校验和（checksum）比较。

7.2.1 行数

计算行数可能是最快也是最简单的确认方法。它只需要在写入结果数据时对数据行进行简单的统计，确认结果记录的数量与预期的数量相匹配。问题是它只在一个维度上确定保真度，它验证了输出行数是否与预期的行数相匹配，但并没有验证记录中的内容。有时候行数是匹配的，但由于转换失败，导致其中一列为空，或者一个数被四舍五入，丢失了精度。因此，可以将这种方法视为一种初查，不能只依赖它来验证数据完整性。

7.2.2 唯一计数

唯一计数是指计算每一列不同单元格的值，可以把它想象成表中每一列的单词计数。因此，如果有一个类似于下面这样的表，那么第 2 列的唯一计数将是 Dog:3 和 Cat:4。

```
1 Dog Foo
2 Dog Foo
3 Cat Foo
4 Dog Foo
5 Cat Foo
6 Cat Foo
7 Cat Foo
```

这种方法主要有 3 个问题。首先，它无法验证唯一值的顺序。因此，如果对上面的表和下面的表执行唯一计数查询，结果是匹配的，但并没有保持保真度，因为第 3 行的 Dog 和第 4 行的 Cat 掉换了顺序。

```
1 Dog Foo
2 Dog Foo
3 Dog Foo
4 Cat Foo
5 Cat Foo
6 Cat Foo
7 Cat Foo
```

其次，它容易受到**高基数列**的影响。高基数列是指包含大量唯一值的列。在前面的示例中，第一列是非重复 ID。这在计算大型数据集时成本非常高。

最后，与检查校验和相比，唯一计数的成本总体上会更高。而且，校验和提供了更高的保真度。

虽然唯一计数不是检查完全保真度的理想选择，但作为初始的完整性检查还是很有用的。

7.2.3 全字节比较

对数据进行全字节比较的成本最高，步骤如图 7-9 所示。

(1) 从数据源读取数据，并保存为原始形式，将其传递给执行数据转换或重新格式化的服务。
(2) 转换数据并写入目标系统。
(3) 处于转换状态的数据位于目标系统中，就像在正常的摄取路径中一样。
(4) 从目标系统提取数据并再次转换回原始状态。
(5) 将初始的原始数据和还原的原始数据保存在同一个存储系统中。
(6) 使用 Apache Spark 等处理工具执行全字节比较。
(7) 如果不存在差异，说明数据管道保持了数据完整性。

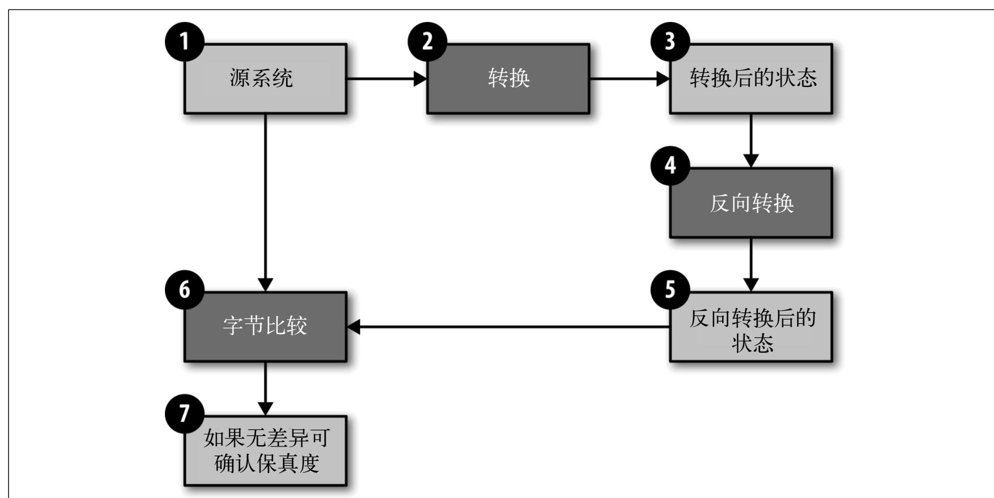


图 7-9：通过全字节比较进行数据验证

重复这个操作，直到相信自己的逻辑是正确的。为了确保持续的数据验证，可以将这个过程自动化，并作为处理流的一部分一直运行下去。这将耗费更多的资源，但如果愿意投入所需的费用和资源，那么就可以在进行常规数据摄取处理的同时大规模运行它。

7.2.4 校验和比较

如果一直进行全字节比较，成本可能会非常高。但即使如此，我们仍然希望保证数据完整性。一种相对简单的方法是使用数据值的校验和。应该怎么做呢？

举一个简单的示例，假设有两张表，一张在关系数据库中，另一张在 Amazon S3 的 Hive 中。表结构如下所示：

```
CREATE TABLE FOO (
 STR_COL STRING,
 INT_COL INT,
 DOUBLE_COL DOUBLE)
```

然后，可以在两张表上运行如下所示的 SQL 语句，确认各列的值是相等的：

```
SELECT SUM(HASH(STR_COL)), SUM(INT_COL), SUM(DOUBLE_COL) FROM FOO
```

你可能会说，如果真实值不一样，那么这个查询仍然可能产生匹配的值。这是有可能的，例如下面的记录：

```
STR_COL  INT_COL  DOUBLE_COL
A        2        1
B        3        2
C        1        3
```

如果担心出现这种问题，可以把 SQL 重写为下面这样：

```
SELECT SUM(HASH(CONCAT(STR_COL, CONCAT(INT_COL, DOUBLE_COL))))) FROM FOO
```

这样一来，校验和针对的是整行记录串联起来的值，可以更好地保证数据的有效性。但是在某些情况下，即使 SQL 返回的值是匹配的，也并不代表实际值是匹配的。这可能是因为散列函数的实现不一样。如果存在这种情况，那么可以尝试使用自定义散列函数。

另外，尽管可能性很小，但是两个数据库的散列函数仍然有可能为相同的值返回不同的散列值。这主要是因为每个系统采用了不同的散列实现。如果存在这种情况，那么可以使用用户自定义函数，从而在两个系统上执行相同的散列逻辑。

7.3 小结

数据完整性是设计和实现数据系统的关键因素。本章讨论了**数据完整性**的含义，并举例说明了如何保持数据完整性和追踪流经系统的数据的谱系。

本章还描述了**完全保真度**与**派生**的含义，然后介绍了两种构建数据管道的方法，基于需求来确保数据完整性。

- **预定义管道**——可以预定义数据在数据管道中的移动路径、处理方式，等等。
- **数据管道的事后发现**——可以在数据通过系统后定义和验证数据谱系。

最后，本章讨论了确保数据保真度的方法，而不管数据是如何在系统中移动的。

第 8 章

数据处理

前文已经讨论了构建数据管道的一些考虑因素，本书的最后一章将讨论如何处理和分析通过这些数据管道收集的数据。围绕收集、存储和管理数据的考虑因素为各种数据架构提供了基础，但只有处理这些数据，才能从数据中获得价值。

与分布式数据架构中的其他组件一样，在处理数据方面所面临的挑战是可用的选项太多，其中的很多选项具有不同的设计目标，并且针对的是不同的用例。本章将像第 5 章一样提供一份区分处理系统的标准清单，从而形成处理系统的评估框架。

处理引擎的选择将取决于用例、经验、团队知识、目标用户、SLA 以及架构组件等因素。希望通过本章的内容能够让你了解不同工具的适用场景，从而在规划项目时做出更明智的决策。

8.1 处理引擎的属性

本章将通过以下属性来区分各种处理引擎。

- 有向无环图管理
  引擎如何处理执行计划？
- 计算隔离
  系统如何分配资源来处理多个用户和作业？

- ❑ 性能

  在不同类型的用例中，作业的执行速度有多快？虽然性能是一个重要的考虑因素，但有时候也需要做出权衡，例如容错。

- ❑ 容错

  引擎是如何对故障做出响应的？作业出现故障时，不同处理引擎的行为会有所不同。在根据具体用例选择合适的引擎时，这些差异就显得非常重要。

- ❑ 交互模型

  用户如何与系统交互？他们是否需要使用 Java 等编程语言来编写代码，或者通过 SQL 这样的声明式模型来与系统交互？

- ❑ 批处理或流处理

  引擎的设计初衷是处理大量记录还是处理单个事件或一小组事件？

接下来，我们将深入探究这些问题，并讨论不同处理引擎的适用范围。先从 DAG 管理开始。

8.1.1 DAG管理

虽然**有向无环图**（directed acyclic graph，DAG）这个术语听起来很吓人，但实际上它是一个非常简单的概念。DAG 是指节点之间的边不存在回路的图。换句话说，图的节点按顺序连接，每个节点都指向序列中的下一个节点。当我们讨论图时，指的不是可视化数值的图，而是由节点和边组成的数学意义上的图。

DAG 类似于查询计划，如关系数据库在编译 SQL 查询时创建的查询计划。分布式引擎为实现预期目标所做的工作可以通过 DAG 和查询计划来表达。

通常，分布式系统提供的 DAG 管理可以通过以下属性来区分：

- DAG 管理是外部的还是引擎内置的；
- DAG 管理是由单个驱动程序还是由多个驱动程序驱动的。

让我们更详细地看一下这些考虑因素。

1. 外部 DAG 管理

这是最简单的实现，因为引擎不需要知道任何与 DAG 有关的信息。大多数现代处理引擎不再使用这种模型，但在 MapReduce 中仍然主要使用它。虽然 MapReduce 仍然是 Apache Hadoop 等项目的核心部分，但已经逐渐被 Apache Spark 这样更现代、更高效的引擎所取代。不过，由于 MapReduce 曾经是实现分布式处理应用程序的主要手段，因此很有必要从历史角度对它进行一番概述，并了解后续系统的谱系。

接下来快速探索一下 MapReduce。如果你已经在分布式数据领域工作了很多年，那么可以跳过下面的内容。

MapReduce

通常，MapReduce 由不同的处理阶段组成，这些阶段被称为 mapper 和 reducer。mapper 从数据源读取数据，对数据进行转换、排序，然后通过一个叫作 shuffle（混洗）的过程对数据进行分区。shuffle 是分布式数据处理的基础部分，它为连接、排序和分组等操作提供了基础。

reducer 位于 shuffle 阶段的另一端，用于处理 mapper 生成的数据分区。reducer 在将数据写入存储系统之前会对分区数据执行转换操作。

图 8-1 描述了这个处理流程：数据输入到 mapper 中，经过处理、排序和混洗，进入 reducer 处理阶段，最后写入存储系统。

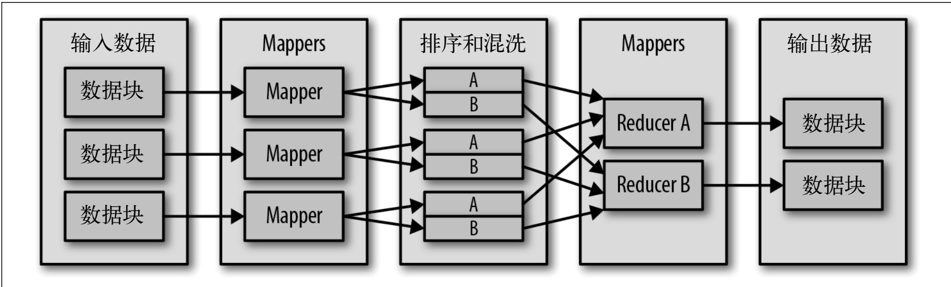


图 8-1：MapReduce 处理流程

虽然 MapReduce 为分布式数据处理提供了一种稳定可靠的工具，但它本身的缺点促使了新引擎的出现。它们逐渐取代了 MapReduce 的位置，成为首选工具。以下是 MapReduce 中最突出的缺点。

- 缺乏 DAG 管理

  如图 8-1 所示，MapReduce 作业的处理顺序是一个 reduce 作业跟着一个 map 作业（但在某些情况下，为了完成某些任务可能会忽略 map 阶段或 reduce 阶段），只能进行单次排序和混洗，然后将数据写回磁盘。如果要做一些更复杂的事情，比如进行两次连接，就需要实现多个 MapReduce 作业，并逐个运行它们。

- I/O 开销的限制

  MapReduce 刚发布的时候，内存还很贵，所以它主要使用磁盘。MapReduce 作业需要在不同的处理阶段从磁盘读取和写入大量的数据。这对作业的性能有很大影响，并最终导致 MapReduce 变慢。

- 启动时间长

  mapper 和 reducer 可能需要很多秒才能启动完毕，在完成任务后，进程将被终止。

- 实现作业有难度

  实现 MapReduce 作业的主要方式是 Java 应用程序，所以这对非程序员来说是一个障碍。此外，如何将问题映射为 MapReduce 作业也很具挑战性。虽然抽象为 MapReduce 提供了更直观的接口，但一般来说，使用 MapReduce 作为处理引擎要求开发人员具备丰富的知识和经验。

- 重量级部署模型

  在部署 MapReduce 作业时，通常需要将大型的 Java Archive（JAR）文件传输到集群节点上。

虽然未来可能不会看到 MapReduce 被用于实现新的处理作业，但在分布式计算中，mapper、shuffle 和 reducer 的模式仍然会反复出现。

MapReduce 曾经非常强大，因为它是当时为数不多可用的分布式执行系统。但由于上述原因，它用起来十分麻烦。这比试图在传统关系数据库上实现 SQL 要复杂得多。

当时，Apache Hive 是针对这些问题的一个解决方案，它提供了 MapReduce 的 SQL 抽象，并支持基于 MapReduce 创建 DAG。虽然 Hive 很慢，而且不适用于交互式查询，但它仍然广泛流行，因为它让 SQL 开发人员也能利用 MapReduce 的强大功能。

2. 内部 DAG 管理

MapReduce 正逐渐被现代处理引擎所取代，如 Apache Spark 和 Apache Flink。这些工具都有一个支持创建 DAG 的执行引擎。这种对 DAG 的支持有助于高效实现复杂的处理作业。

甚至是最初作为 MapReduce 抽象的 Hive 也支持能够创建更复杂的 DAG 的引擎，例如 Apache Tez，它提供了一个支持创建 DAG 的框架。

8.1.2 计算隔离

下一个类别标准是系统如何管理计算隔离，实际上就是指系统如何管理执行任务所需的资源。可以将其分为以下几种隔离级别。

- 节点级别

  一个节点只运行一个作业。

- 容器级别

  一个容器只运行一个作业，但一个节点可以有多个容器。

- 任务级别

  由执行引擎管理进程内的多个任务。

不同的系统支持不同的隔离级别，接下来逐一讨论它们，并在不同的用例中使用它们。

1. 节点级别的隔离

在节点级别，一个节点只能运行一个与作业相关的任务。随着 Amazon Elastic MapReduce（EMR）等云产品的兴起，节点级别的隔离变得越来越流行。在这个模型中，用户将启动一个 EMR 集群来执行作业，然后在作业完成时终止集群。

这样做的优点是不需要担心其他作业会争夺资源。但一个缺点是需要额外的时间来启动集群实例。另一个潜在的缺点是低资源利用率，因为在执行作业时，并非节点的所有资源都能被 100% 利用。

2. 容器级别的隔离

容器级别的隔离具有两个维度：一个是在集群资源管理器和作业调度程序中（如 YARN 和 Mesos）运行容器；另一个是使用 Docker 和 Kubernetes 等解决方案，它们提供了虚拟化服务，允许在单台服务器上运行多个容器。

尽管 Docker 和 Kubernetes 等容器技术是部署架构中的集成组件，但它们不会影响你对处理引擎的选择，因为这些技术与部署在它们内部的软件无关。

不过，处理引擎的选择可能与集群资源管理器有着紧密的联系。例如，为了能够管理资源，Spark 和 MapReduce 等引擎通常会在 YARN 或 Mesos 等系统中运行。在 YARN 中运行 Spark 时，YARN 可以控制将任务分配给可用的容器。

3. 任务级别的隔离

很多现代处理引擎（例如 Apache Impala、Apache Presto 和 Apache Spark）支持任务级别的隔离，可以在单个系统进程中运行不同的工作负载。这意味着进程需要在任务之间切换，并且可能会产生资源冲突。但这样可以提高作业效率，例如在执行任务时不需要启动进程。

4. 隐藏的隔离

还需要指出的是，在某些情况下，隔离被隐藏起来了。在使用某些基于云的工具时就是这样，如 Amazon Athena 和 Amazon Lambda。这也被称为**无服务器计算**。

5. 隔离级别的考虑因素

选择隔离类型取决于如下因素。

- 作业数量与预算

  如果公司有充足的资金，并且可以接受节点级别的隔离，那么这通常最容易做出选择。

- 有限的支出

  如果预算有限，可能需要采用多租户架构，也就是容器级别的隔离。例如，使用 YARN 运行多个作业的 Hadoop 集群。

❑ 严格的延迟需求

如果要求低延迟，那么支持任务级别隔离的系统可能是最佳选择。

❑ 云部署

如果系统部署在云端，那么在使用特定的云服务时，它们可能已经为你选择好了隔离级别。

8.1.3 性能

尽管引擎在执行作业时的性能非常重要，但我们不能只看那些能够执行特定工作负载的引擎，虽然它们的速度很快。更重要的是了解工具在执行用例时具备怎样的性能。在某些情况下，工具执行查询的速度可能很快，但涉及多个连接的复杂查询可能会存在性能问题。来看看下面这些工具的属性。

❑ Presto

Presto 提供了一个速度很快的执行引擎，但如果查询超过了内存限制就会出现问题。

❑ Spark

Spark 很快，但目前不支持在多租户环境中使用外部工具（如商业智能工具）。

❑ 带有 Tez 引擎的 Hive

带有 Tez 引擎的 Hive 速度很快，但仍然需要处理 Hive 架构中的其他组件，如 Hive Metastore。

此外，对工具的选择可能会受到某些因素的驱动，比如决定将作业迁移出已有的系统。你可能在 Teradata 系统中部署了现有的数据模型和查询。有很多 Teradata 应用程序使用了基于索引列的本地连接，可以基于单个连接键快速连接多个表。如果这个数据模型已经使用了多年，并且基于它构建了数千个作业，那么就很难将它迁移到不提供类似优化的执行引擎。在这种情况下，Hive 的分桶排序连接可以作为本地索引的替代方案，而其他引擎可能无法支持这种模型。重要的是，你对工具的选择不仅受到原始性能的影响，还取决于实际用例，所以最好先根据特定工作负载运行测试。

8.1.4 容错

通常，处理框架可以分为以下 3 种容错级别：

- 零容错；
- 执行器恢复；
- 完全作业恢复。

接下来详细介绍每一种级别。在讨论分布式系统的容错时，会反复提及如何在可恢复性和性能之间做出权衡。

1. 零容错

要讨论的第一个级别是零容错，换句话说，就是在作业发生故障时不提供恢复选项。这听起来似乎不太好。但实际上，如果优先考虑作业的性能，那就可以这样做。没有恢复方面的开销意味着引擎可以专注于提升查询的执行速度，这对临时查询和分析查询来说是有利的，因为这些查询通常会在执行完成后快速向用户返回结果。即使发生了故障，这些引擎通常也能够足够快地重新启动查询，并快速返回结果。

但是，这些工具不太适合长时间运行的任务，例如 ETL 作业，不过这些工作负载类型并不是这些引擎的目标。

Impala 和 Presto 就属于这类工具。这些引擎旨在有效地提供分析查询，非常适合与商业智能和分析工具集成在一起。Impala 和 Presto 的典型查询可以在几秒内完成。在这个时间范围内发生节点故障的概率非常低。即使发生节点故障，重新处理查询的时间也很短，不会有什么问题。

不过需要注意的是，容错引擎（如 Spark SQL）的性能已经有所提升。尽管 Spark SQL 的性能可能与 Impala 和 Presto 不同，但几乎可以满足对性能的需求，并能够在发生故障时提供恢复机制。请注意，在撰写本书时，Spark SQL 架构还存在一些缺陷，不太适合用在多用户环境的商业智能和分析工具中。

2. 执行器恢复

Hive、MapReduce 和 Spark 等系统就属于这个类别。只要驱动程序进程没有丢失，那么这类系统就会在丢失执行程序进程时提供恢复机制。换句话说，你可能会丢失一个或多个正在执行作业的任务，但只要驱动程序进程仍在运行，作业就不会失败。

以一个长期运行的作业为例，如 *k* 均值聚类作业。执行 *k* 均值聚类算法的典型作业将对数据进行多次迭代。这可能需要几十分到几小时才能完成，具体取决于迭代次数、维度数量和数据规模。显然，如果作业已经运行了几分或几小时，而且在发生故障时必须完全重启，那么成本会非常高。所幸的是，有了这种恢复机制，系统可以简单地将失败任务移动到新进程中，并重新执行这个任务。

图 8-2 显示了一个 Spark 作业示例，此时执行器发生了故障。在这种情况下，仍然可以使用已完成任务的输出结果，系统只需要重新执行失败的任务和其他必要的任务（作为失败任务的输入）。

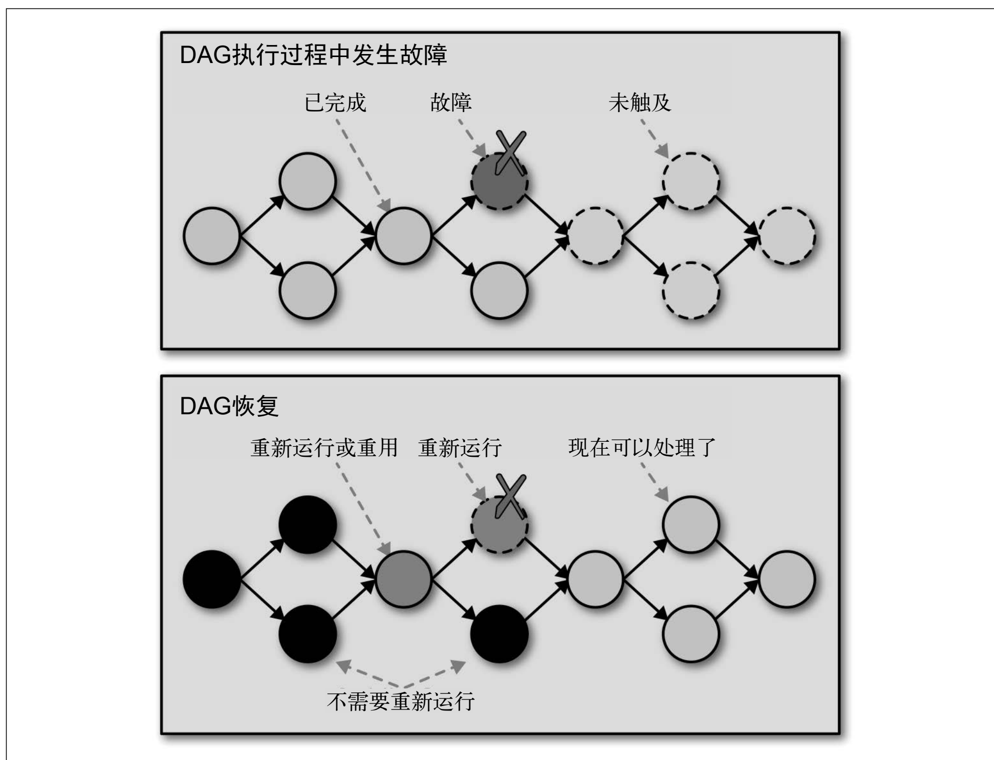


图 8-2：支持执行器恢复的系统提供的恢复机制

这种类型的恢复机制增加了处理引擎的复杂性，但对长期运行的作业来说是有利的，当这些作业需要满足指定的 SLA 时尤为如此。

执行器的恢复也可能存在局限性。以 Spark 为例，因为处理状态是由驱动程序负责维护的，所以，如果没有高可用性配置，一旦驱动程序进程丢失，作业也就跟着丢失，需要重启并从头开始执行作业。

3. 完全作业恢复

完全作业恢复机制在流处理引擎中很常见。这类引擎包括 Flink、Spark Streaming 和 Kafka Streams。流处理作业需要保持长时间运行（数周甚至数月），可想而知，这种恢复机制有多重要。在这样长的运行时间里，故障是不可避免的，包括与驱动程序相关的故障。

这些系统使用的恢复机制已经超出了本章的讨论范围，但总体来说，这些引擎通过检查点之类的机制来支持恢复。这确实需要额外的开销，比如用来存储检查点的持久存储，以及读写检查点的开销。

8.1.5 交互模型

通常，与执行引擎的交互通过代码、SQL 或两者的组合来完成。可以将工具分为以下几类。

- JDBC 或 ODBC 访问

  可以像普通关系数据库一样使用 SQL 访问系统，并使用第三方的商业智能和分析工具与引擎交互。大多数支持 SQL 接口的工具也支持这种类型的访问，例如 Hive、Impala、Presto、Cassandra 和 Spark SQL（不过之前提到了它的局限性）。

- SQL 代码

  与标准关系数据库类似，可以执行 SQL 查询。上一个类别中的工具几乎也属于这一类别。这些引擎中的大多数还支持用户定义函数扩展。用户定义函数可以执行引擎非原生支持的复杂查询。

- 非 SQL 声明式模型

  一些工具提供了类似 SQL 的声明式编程接口。这些接口可以用来表达复杂的编程任务，但比编写代码要简单一些。Apache Pig 就是这种模型的一个很好的示例，它提供了一种叫作 Pig Latin 的声明式编程语言。尽管这些引擎对于某些用例的实现非常有用，但随着 Spark 等引擎的出现，这些工具的使用量似乎也在下降，这也可能意味着这方面的资源和开发经验也在减少。因此，最好仔细评估这些工具在新项目中的使用情况。

- 编程代码

  这些引擎要求开发人员使用 Python、Java 或 Scala 等编程语言来实现处理逻辑。Spark、MapReduce 和 Flink 就属于这一类。

由于某些属于编程代码类别的引擎可以与更高级别的抽象（如 SQL 抽象）一起使用，因此情况就变得有点复杂。例如，Hive 实际上可以将查询编译为其他执行引擎（如 MapReduce 或 Spark）可以执行的代码。

决定使用哪种引擎取决于用例和团队的经验。如果是分析师在处理，那么可以考虑使用能够提供 SQL 接口的引擎，比如 Hive。那些无法使用 SQL 轻松完成的作业就可以使用 Spark 或 Flink 等编码引擎来开发。

8.1.6 批处理和流处理

最后一个类别与工作负载类型有关。一般来说，可以将其分为适用于批处理作业的引擎和适用于流处理作业的引擎。批处理引擎定期或一次性处理大量记录的作业。这些作业通常需要几分到几小时才能完成。流处理引擎则旨在持续地处理传入事件，并且通常会在更短的时间内产生结果。

MapReduce 是批处理引擎的一个典型示例，它适用于需要处理大块记录的作业。之后出现

了 Spark，虽然它比 MapReduce 更高效，但仍然以长时间运行的批处理作业为目标。

流处理引擎旨在实现更加实时的数据处理。这些引擎对传入的事件流执行持续的处理，并以秒级或亚秒级的速度返回结果。这类引擎包括 Spark Streaming、Kafka Streams、Apache Storm 和 Apache Heron。

你可能已经注意到，Spark 在批处理和流处理两种类别中都有提及。这是因为 Spark 处理模型同时支持批处理和流处理，尽管在实现和部署方面存在一些小差异。此外，Flink 也同时支持这两种模型，并且同样有一些小差异。使用支持两种模型的工具有如下好处：

- 只需要学习一个编程接口就可以实现批处理和流处理；
- 可以在批处理作业和流处理作业之间重用代码；
- 降低部署和管理应用程序的复杂性。

是选择批处理还是流处理，完全取决于具体的用例。对于 ETL 或机器学习应用程序，选择批处理总是没错的。流处理适用于需要对传入数据做出几近实时反应的作业，例如，欺诈检测系统和需要基于数据窗口运行计算的应用程序。

需要注意一些典型的流处理引擎特性，其中一个是处理流程的实现方式。Apache Storm 及其后继的 Apache Heron 等工具采用了**拓扑模型**。与 DAG 类似，拓扑模型是一种抽象，用于表示数据转换图。Spark 就使用了 DAG 模型。主要区别在于如何使用不同的模型来编写应用程序代码。我们可以使用这两种模型来表达大多数的问题，所以这可能不是选择处理引擎时的主要考虑因素。

另一个特性是处理状态的管理方式。Storm 等工具在内存中管理处理状态，如果要进行持久化状态管理，需要做一些额外的工作。Spark 等工具为状态管理提供了更好的支持，如果进程丢失，可以进行故障恢复，相关的数据还保留在内存中。

8.2　数据处理演变史

可以从某个角度来回顾一下数据处理管理系统在过去几十年的演变过程。某些细节可能存在争议，不过图 8-3 很好地说明了这种演变。从数据管理的角度来看，演变过程包括以下方面。

- 关系数据库（单节点 RDBMS）在 20 世纪 70 年代开始出现。这些早期的数据库通常在单台服务器上运行，并且只管理这些服务器可容纳的数据量。
- 因为需要管理更大的数据量，所以出现了分布式数据库。它们以大规模并行处理（massively parallel processing，MPP）数据库为代表，如 Teradata 和 Vertica。
- 随着数据量和数据类型的不断增长，不管是在技术层面还是在经济层面，MPP 系统对于很多工作负载来说都已经不切实际了，于是出现了新的数据管理系统，如 Hadoop、Cassandra 等。

随着数据管理和存储方式的变化，数据处理方式也在变化。

- 数据集成需求（比如移动和转换数据）带来了 ETL 工具的发展，这些工具被用来实现和自动化数据集成流程。
- 与数据存储一样，随着时间的推移，数据量和数据源种类在增加，因此，出现了新框架来执行高效的数据并行处理，例如 MapReduce 和 Spark。这些框架将 ETL 处理从传统的数据管理系统和专有工具转移到了新的开源分布式系统上。

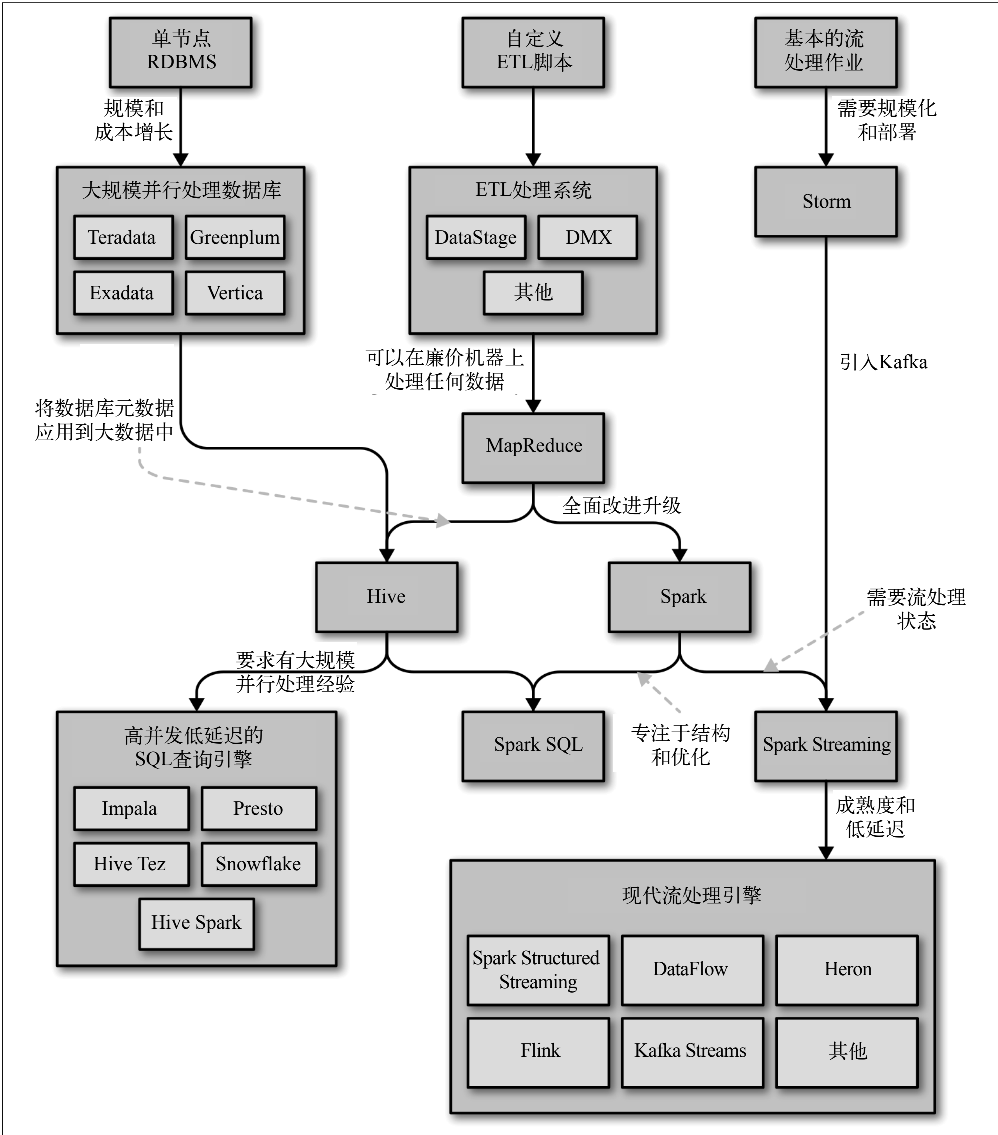


图 8-3：数据处理的演变

流处理系统的出现和演化也是开源数据管理工具发展的一部分。Apache Storm 和之后的 Spark Streaming 都是早期的例子。如今有很多功能强大的工具可用于执行几近实时的流处理，而这在以前只有昂贵的专有解决方案才能做到。

在此期间，可靠且有效的存储和转换一直是常态需求。与现在相比，最大的不同是规模上的巨大变化，我们已经从单节点数据库发展到可以支持数千个节点并且每天处理数百万到数十亿条记录的系统。这种规模在几十年前是不可想象的。

此外，我们已经从只有几个主要数据管理软件供应商的世界进入了拥有大量选择的世界，其中包括像 Oracle 和 IBM 这样的大型供应商，以及其他的开源项目和供应商。云供应商的出现进一步拓宽了数据管理的版图。

8.3 小结

本章提供了一些指南，用于根据并行处理引擎的特征来评估处理框架。这些特征如下所述。

- 引擎对执行计划定义的支持，特别是在实现应用程序时是否支持 DAG 定义。在实现更复杂的处理流程时，支持 DAG 定义是一个重要的考虑因素。
- 执行引擎的隔离模型。这可能会影响工具的性能，在多租户环境中尤为如此。
- 工具的性能特征。性能不只是指引擎的原始速度。例如，某个特定引擎可以快速地运行查询，但不提供很好的容错机制。
- 在性能和容错的权衡方面，不同的引擎提供了不同的机制来处理作业故障。当作业出现部分故障时，某些工具为作业继续运行提供了强大的支持，而其他工具需要完全重新启动作业。
- 用户与引擎的交互方式。他们可以直接执行 SQL 查询，还是需要编写代码？这个考虑因素很重要，具体取决于团队成员的背景和掌握的技能。
- 工具是否为批处理或流处理（或两者兼有）而设计。一般来说，这两个模型适用于不同的用例。因此，在选择引擎来实现特定的用例时，这是一个重要的考虑因素。

总体来说，除了上述考虑因素，处理引擎的选择在很大程度上也会受到团队技能的影响。

- 如果拥有一个分析师团队，他们需要高效地执行分析查询，那么像 Impala 或 Presto 这样的工具可能是不错的选择。
- 如果开发人员可以熟练使用 SQL 实现 ETL 作业，那么需要一个支持容错且基于 SQL 的引擎，因此 Hive 或 Spark SQL 可能比 Impala 或 Presto 更合适。
- 如果 Java 或 Scala 开发人员需要实现复杂的机器学习算法，那么最好选择可以实现复杂批处理作业的工具，如 Spark 或 Flink。

希望你能够将本章提供的指南与用例相结合，找到合适的工具来满足数据项目的处理需求。

关于作者

特德 · 马拉斯卡（**Ted Malaska**）是 Capital One 的企业架构主管。在加入 Capital One 之前，他在暴雪娱乐公司担任全球视野工程总监，负责为《魔兽世界》《守望先锋》和《炉石传说》等游戏提供支持。特德曾经是 Cloudera 首席解决方案架构师，帮助客户在 Hadoop 生态系统中找到成功的解决方案。他还曾经是美国金融业监管局的首席架构师。他为 Apache Flume、Apache Avro、Apache Yarn、Apache HDFS、Apache Spark、Apache Sqoop 等开源项目贡献过代码。特德是《Hadoop 应用架构》的合著者，经常在技术大会上发表演讲，也经常撰写有关数据架构的文章。

乔纳森 · 塞德曼（**Jonathan Seidman**）是 Cloudera 云计算团队的软件工程师。在此之前，他是 Cloudera 的解决方案架构师，与合作伙伴一起将他们的解决方案与 Cloudera 的软件栈集成在一起。在加入 Cloudera 之前，他是 Orbitz Worldwide 大数据团队的技术负责人，负责为一个流量巨大的网站管理 Hadoop 集群。他也是芝加哥 Hadoop 用户组和芝加哥大数据线下活动的联合创始人、《Hadoop 应用架构》合著者、《Hadoop 硬实战》技术编辑，并曾经在与 Hadoop 和大数据相关的多个行业大会上发表演讲。

关于封面

本书封面上的动物是白头牛文鸟（*Dinemellia dinemelli*）和黄胸织布鸟（*Ploceus philippinus*），它们属于文鸟科。这一科的鸟类俗称“织工”，因为它们用树枝和树叶纤维等天然材料编织巢穴。

白头牛文鸟生活在非洲，它们习惯以被非洲水牛惊扰的昆虫为食，并因此而得名。除了昆虫之外，白头牛文鸟还以水果和种子为食。它们成群结队地觅食，具有很强的群居属性。

黄胸织布鸟生活在印度和东南亚，它们在树枝上筑巢，巢呈管状。和白头牛文鸟一样，黄胸织布鸟也会成群结队地觅食。它们以大米等谷物为食，因此在印度部分地区被列为农业害虫。

O’Reilly 图书封面上的很多动物濒临灭绝，它们是这个世界所剩无几的瑰宝。如果想了解如何为这些动物提供帮助，请访问 animals.oreilly.com。

白头牛文鸟的图片来自 *Wood’s Animate Creation*，黄胸织布鸟的图片来自 *Cassell’s Natural History*。



微信连接



回复“大数据”查看相关书单



微博连接

关注@图灵教育 每日分享IT好书



QQ连接

图灵读者官方群I：218139230
图灵读者官方群II：164939616

图灵社区
iTuring.cn

在线出版，电子书，《码农》杂志，图灵访谈

O'REILLY®

大数据项目管理 从规划到实现

许多公司会在大数据项目的实施细节上下很多功夫，例如研究分布式处理引擎和数据分析算法。这并没有错，但不要因为一棵树而错过整片森林。本书将为你打开更广阔的视野，展示如何从大数据项目的规划阶段开始，一步步走向成功。无论是首席信息官、首席技术官、项目经理，还是架构师和开发人员，都能通过本书得到启迪。

- 开始规划：思考大数据项目的主要类型
- 评估和选择数据管理解决方案
- 降低与技术、团队、需求相关的风险
- 探索良好的接口设计模式
- 为项目选择合适的分布式存储系统
- 规划和实施元数据收集
- 使用数据管道确保数据完整性
- 根据并行处理引擎的特征评估处理框架

特德·马拉斯卡（Ted Malaska），Capital One的企业架构主管，曾在暴雪娱乐公司担任全球视野工程总监，负责为《魔兽世界》《守望先锋》《炉石传说》等游戏提供支持。他为众多开源项目贡献过代码，并与塞德曼等人合著有《Hadoop应用架构》。

乔纳森·塞德曼（Jonathan Seidman），Cloudera云计算团队的软件工程师。在加入Cloudera之前，他是Orbitz Worldwide大数据团队的技术负责人，负责为一个流量巨大的网站管理Hadoop集群。塞德曼与马拉斯卡等人合著有《Hadoop应用架构》。

封面设计：Karen Montgomery 张健

图灵社区：iTuring.cn
热线：(010)51095183转600

分类建议 计算机/大数据与云计算

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版





ISBN 978-7-115-45736-3
定价：59.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这可以找到我们：

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks